



RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

AACHEN INSTITUTE FOR ADVANCED STUDY IN
COMPUTATIONAL ENGINEERING SCIENCE

MASTER'S THESIS

Hierarchical Performance Modeling for Ranking Dense Linear Algebra Algorithms

Elmar Peise

May 4th, 2012

SUPERVISOR
Paolo Bientinesi

CO-EXAMINER
Martin Bucker

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Aachen, May 4th, 2012

Elmar Peise

Contents

1	Introduction	1
1.1	Motivating Example	1
1.2	Related Work	3
1.3	Outline of this Thesis	3
1.4	Blocked Algorithms	4
1.4.1	Triangular Inverse $L \leftarrow L^{-1}$	4
1.4.2	Generalization	5
2	Sampling	9
2.1	The Goal	9
2.1.1	Performance Metrics	9
2.1.2	Routines and Arguments	11
2.1.3	Systems and Environment	11
2.2	Experiments	11
2.2.1	Reference Experiment	12
2.2.2	Modified Setups	13
2.3	The Sampler	15
2.3.1	Design	15
2.3.2	Interface and Usage	16
2.3.2.1	Configuration File	16
2.3.2.2	Input Stream Format	17
2.3.2.3	Output Stream Format	18
3	Modeling	19
3.1	Preliminary Experiments	19
3.1.1	Discrete Arguments	20
3.1.2	Size Arguments	21
3.1.3	Scalar Arguments	23
3.1.4	Vector and Matrix Arguments	24
3.1.5	Leading Dimension and Increment Arguments	24
3.2	The Targeted Models	25
3.2.1	Creation and Reasoning	25
3.2.2	Evaluation of a Model at a Point	27
3.3	The Modeler	28
3.3.1	Sampler Interface	29
3.3.2	Routine Modeler	30
3.3.2.1	Generation of Sampling Requests	30

3.3.2.2	Processing Sampling Results	32
3.3.2.3	Assembly of the Model	32
3.3.3	Piecewise Polynomial Modelers	32
3.3.3.1	Polynomial Fitting through Least Squares	33
3.3.3.2	Approximation Accuracy	34
3.3.4	PModeler: Model Expansion	34
3.3.4.1	Initial Model	35
3.3.4.2	Region Expansion	35
3.3.4.3	Region Generation	37
3.3.5	PModeler: Adaptive Refinement	40
3.4	Results	41
3.4.1	<i>flops</i>	42
3.4.1.1	Model Expansion.	42
3.4.1.2	Adaptive Refinement	42
3.4.2	<i>ticks</i>	42
3.4.2.1	Model Expansion	43
3.4.2.2	Adaptive Refinement	45
3.4.2.3	Comparison	47
4	Prediction and Ranking	49
4.1	Representation of Blocked Algorithms	49
4.2	Triangular Inverse $L \leftarrow L^{-1}$	51
4.3	LU Decomposition $LU \leftarrow A$	54
4.4	Sylvester Equation: Solving $LX + XU = C$ for X	55
5	Conclusion and Future Work	61
5.1	Outlook	62
	Bibliography	65
	List of Figures	67
A	Introduction to BLAS	69
A.1	Routine names	69
A.2	Arguments and Matrix Storage	71
A.3	Routine Invocation and Results	72
A.4	Implementations	72
B	Implementations of Blocked Algorithms	75
B.1	Triangular Inverse $L \leftarrow L^{-1}$	75
B.2	LU Decomposition $LU \leftarrow A$	78
B.3	Sylvester Equation: Solving $LX + XU = C$ for X	82

Chapter 1

Introduction

A large class of dense linear algebra operations, such as LU decomposition or inversion of a triangular matrix, are usually performed by blocked algorithms. For one such operation, typically, not only one but many algorithmic variants exist; depending on architecture, libraries, and problem size, each variant attains a different performance. Our goal is to rank the algorithmic variants according to their performance for a given scenario *without* executing them.

For this purpose, we analyze the routines upon which the algorithms are built and introduce a tool that, based on measurements, models their performance. The generated performance models are then used to predict the performance of the considered algorithmic variants. For a given scenario, these predictions allow us not only to rank the variants but to determine the optimal algorithmic block-size.

The strength of our approach mainly originates from the performance models. Generated once for a given system, they can be used to analyze any number of blocked algorithms. Besides reliably predicting and ranking algorithm performance, they yield further insight into how the performance is influenced by certain factors such as the block-size.

1.1 Motivating Example

We consider an exemplary dense linear algebra operation: the inversion of a lower triangular matrix, $L \leftarrow L^{-1}$. For this operation, there exist four different blocked algorithms (see [Section 1.4](#)). For now, all we need to understand is that all of them take a lower triangular matrix $L \in \mathbb{R}^{n \times n}$ as an input and compute its inverse in place; they accept only one additional algorithm parameter: the algorithmic block-size b .

We use the following setup to analyze the four algorithms:

- Their implementation (see [Appendix B.1](#) for the source code) is compiled with Intel's C Compiler (`icc`) version 12.0 [\[4\]](#).
- They are executed on one core of an Intel Harpertown E5450 processor [\[6\]](#) running at 2.99GHz. This processor can issue 2 double precision floating point operations¹ per clock cycle. Therefore, it can perform up to $peak_flops/s = 2 \times 2.99 \cdot 10^9$ floating point operations per second.

¹ We consider a fused multiply add operation (FMA) $d \leftarrow a \cdot b + c$ to be *one* floating point operation. It is the core of any dense matrix operation.

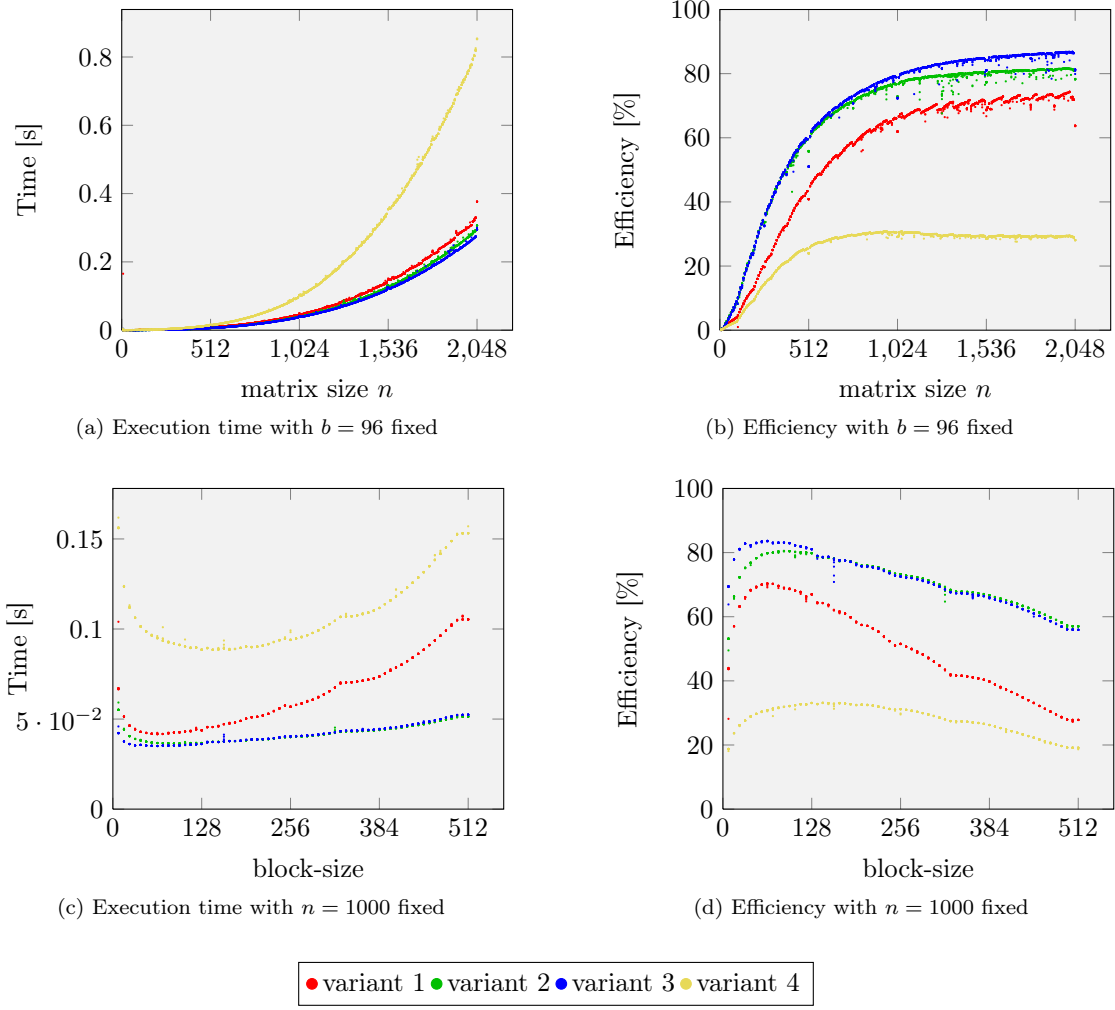


Figure 1.1: Inversion of a triangular matrix: execution time and efficiency.

- The Intel's Math Kernel Library (MKL) [5] is used for the underlying Basic Linear Algebra Subroutines (BLAS) [20, 13].

Figure 1.1 shows the execution time and the efficiency² of the four algorithm variants. In Figures 1.1a and 1.1b, the algorithmic block-size is fixed to 96 and the matrix size n is varied. The results show significant differences in performance between algorithms: Variant 4 (●) takes more than twice as long compared to the other three variants; it reaches a maximum efficiency of 29%. Variant 1 (●) is also slower than variants 2 (●) and 3 (●) and attains an efficiency of 73%. Variants 2 (●) and 3 (●) seem equally good for matrices up to size $n = 512$. For larger matrices, variant 2 (●) becomes the fastest; ultimately, variants 2 (●) and 3 (●) reach efficiencies of 81% and 86%, respectively.

In Figures 1.1c and 1.1d, the matrix size is fixed to $n = 1000$ and only the block-size varies.

² Section 2.1.1 contains details on the computation of efficiency.

For all variants, the efficiency decreases for very small and large block-sizes. Variants 1 (●), 2 (●), and 3 (●) reach their peak efficiency for block-sizes close to 100.

This example shows that in order to reach high efficiency, it is crucial to both choose the algorithmic variant as well as optimizing the block-size. Due to the complexity of the architecture and the memory access patterns, it is not possible to determine the optimal configuration by only analyzing the algorithms mathematically. On the contrary, the best choice depends on the matrix size, the underlying computational kernels, such as BLAS³, and the processor architecture; changing these may lead to entirely different performance behavior.

In this thesis, we introduce tools and methods to analyze and model the performance of dense linear algebra kernels. These tools allow us to perform the challenging task of ranking algorithmic variants according to their performance and determining the optimal block size. Results for the example at hand are given in [Section 4.2](#).

1.2 Related Work

Several different approaches of using performance modeling in dense linear algebra already exist.

Iakymchuk et al. [19] model the performance of BLAS [20, 13] analytically based on memory access patterns. While their models represent the program execution very accurately, constructing them requires a high level of expertise on both the routines and the architecture.

Cuenca et al. [12] develop a self-optimizing linear algebra routines (SOLAR). In their system, every routine is associated with performance information, which is hierarchically propagated to higher level routines (e.g., BLAS → LAPACK → ScaLAPACK); on each level, the information is used to tune the routines and associate according performance information.

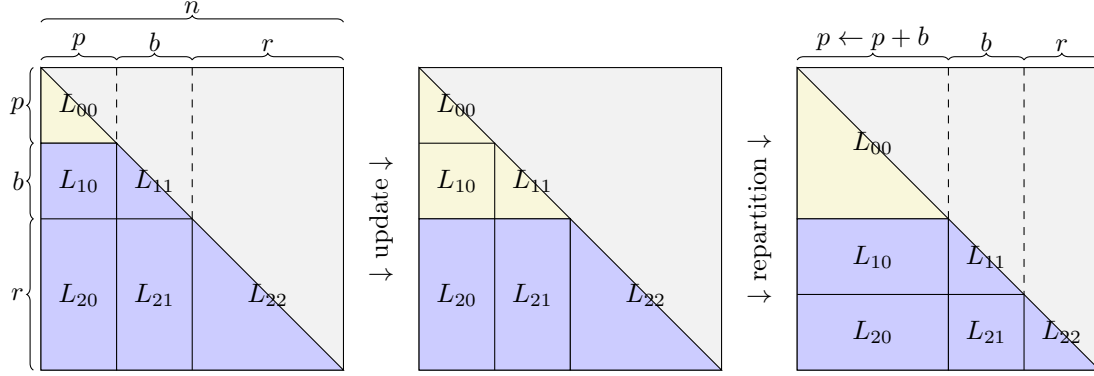
Dongarra et al. [14] propose a modeling approach targeted at programs such as High Performance LINPACK (HPL) [22] and ScaLAPACK [11, 9]. They employ sampling and polynomial fitting to construct their models and use them to extrapolate the performance of routines for larger problems and higher parallelism.

1.3 Outline of this Thesis

This thesis is structured as follows:

- In [Section 1.4](#), we give an introduction to blocked algorithms, the target of our predictions.
- In [Chapter 2](#), we discuss the *Sampler*, a tool that measures different performance metrics during the execution of dense linear algebra routines.
- In [Chapter 3](#), we introduce the *Modeler*, a framework that, based on the Sampler, generates analytical performance models for dense linear algebra routines.
- In [Chapter 4](#), we use the performance models generated by the Modeler to predict the performance of blocked algorithms and rank them accordingly.
- In [Chapter 5](#), we conclude with a summary of our achievements and an outlook on possible future research directions based on this thesis.

³ An introduction to BLAS is given in [Appendix A](#).

Figure 1.2: Triangular Inverse — traversal of L .

1.4 Blocked Algorithms

Since our goal is to rank blocked algorithms, in this section, we introduce their structure. We begin by studying blocked algorithms for a specific operation, the inversion of a triangular matrix (1.4.1), and then generalize the concepts to arbitrary operations (Section 1.4.2).


1.4.1 Triangular Inverse $L \leftarrow L^{-1}$

In Section 1.1, we discussed the performance of four blocked algorithms for the inversion of a lower triangular matrix. The algorithms take a lower triangular matrix $L \in \mathbb{R}^{n \times n}$ as an input and compute its inverse in place.

Within a blocked algorithm, the matrix L is seen in a *partitioned* form

$$L = \begin{pmatrix} L_{TL} & 0 \\ L_{BL} & L_{BR} \end{pmatrix},$$

where $L_{TL} \in \mathbb{R}^{p \times p}$, $L_{BL} \in \mathbb{R}^{q \times p}$, and $L_{BR} \in \mathbb{R}^{q \times q}$ with $p + q = n$. Initially, p is 0, or equivalently,

L_{TL} and L_{BL} are "empty". L is then traversed from the top left corner along its diagonal  in steps of the algorithmic *block-size*, increasing p up to n . During the traversal, the inverse of L is computed in L_{TL} ; this part of the matrix grows in size until $p = n$. At this point, L_{TL} is of the size $n \times n$ and contains the inverse of the original matrix L ; the algorithm terminates.

Traversing L not element-wise but in blocks of size b has the advantage that BLAS Level-3 operations, such as `dgemm`, can be used to attain high performance.

At each step of the matrix traversal (depicted in Figure 1.2), L is repartitioned as

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|cc} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{array} \right),$$

with $L_{11} \in \mathbb{R}^{b \times b}$, $L_{22} \in \mathbb{R}^{r \times r}$ (with $r = n - p - b$), and conforming sizes for the other submatrices. (When n is not divisible by b , b is adjusted to $b \leftarrow n - p$ in the last step.)

At this point, the four algorithms perform different updates on the 3×3 partitioned form of the matrix:

Variant 1	Variant 2
$L_{10} \leftarrow L_{10}L_{00}$ $L_{10} \leftarrow -L_{11}^{-1}L_{10}$ $L_{11} \leftarrow L_{11}^{-1}$	$L_{21} \leftarrow L_{22}^{-1}L_{21}$ $L_{21} \leftarrow -L_{21}L_{11}^{-1}$ $L_{11} \leftarrow L_{11}^{-1}$
Variant 3	Variant 4
$L_{21} \leftarrow -L_{21}L_{11}^{-1}$ $L_{20} \leftarrow L_{21}L_{10} + L_{20}$ $L_{10} \leftarrow L_{11}^{-1}L_{10}$ $L_{11} \leftarrow L_{11}^{-1}$	$L_{21} \leftarrow -L_{22}^{-1}L_{21}$ $L_{20} \leftarrow -L_{21}L_{10} + L_{20}$ $L_{10} \leftarrow L_{10}L_{00}$ $L_{11} \leftarrow L_{11}^{-1}$

All but the last update statement in each algorithm map directly to one of the BLAS routines `dtrsm`, `dtrmm`, and `dgemm` (see [Appendix A](#) for an introduction to BLAS). The last update on the other hand is the inversion of $L_{11} \in \mathbb{R}^{b \times b}$ — a recursive call to the inversion of a triangular matrix with a smaller input matrix of size $b \times b$. This recursive invocation, which is typical for blocked algorithms, is performed by an *unblocked* version of the algorithm (that is block-size $b = 1$). In this version, the last update is the scalar operation $L_{11} = \frac{1}{L_{11}} \in \mathbb{R}$.

Once the updates have been performed, the 3×3 partitioning of the matrix L is merged back into four quadrants

$$\left(\begin{array}{cc|c} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array} \right) \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right)$$


and p is incremented by b . Unless $p = n$, the matrix is now again partitioned into

$$\left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|cc} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array} \right)$$

and the matrix traversal proceeds. When $p = n$, the algorithm terminates and L^{-1} has been computed.


1.4.2 Generalization

In the previous section, we presented four blocked algorithms for the inversion of a lower triangular matrix. We now generalize the principle of blocked algorithms step by step.

Full matrix. Extending blocked algorithms to non-triangular matrices is straightforward. Assuming the same direction of traversal , an input matrix A is partitioned as follows:

$$A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right).$$

The corresponding update and repartitioning are shown in [Figure 1.3](#).

Traversal directions. Until now, we considered the case where the input matrix A is traversed from the top left corner along its diagonal . In principle, A can be traversed in any direction:

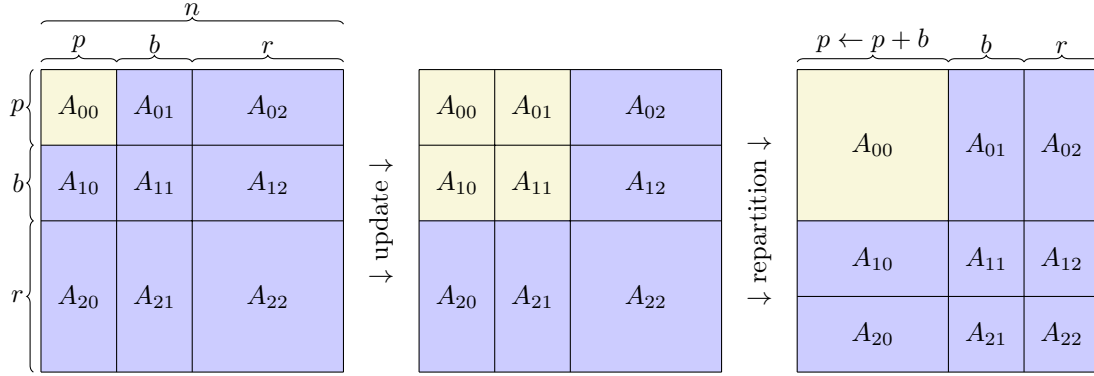


Figure 1.3: Blocked algorithms — update and repartitioning for a square matrix.

In all cases that traverse the matrix diagonally, it is partitioned into 2×2 and 3×3 as shown above. In the other cases, the matrix is partitioned into two and then three matrices

$$A \rightarrow \begin{pmatrix} A_T \\ A_B \end{pmatrix} \rightarrow \begin{pmatrix} A_0 \\ A_1 \\ A_2 \end{pmatrix} \text{ or } A \rightarrow \begin{pmatrix} A_L & A_R \end{pmatrix} \rightarrow \begin{pmatrix} A_0 & A_1 & A_2 \end{pmatrix}$$

and traversed in blocks of size b .

Multiple matrices. When a blocked algorithm operates on multiple matrices, each of them can potentially be traversed along different directions. The block-size b , however, is constant across all matrices.

Non-square matrices. When non-square matrices are traversed horizontally or vertically, nothing changes compared to the square case. When, on the other hand, a non-square matrix is traversed diagonally, there are two alternatives:

- Using two different block-sizes, leading to rectangular blocks on the diagonal, or
- Traversing diagonally as far as possible and continuing along the remaining vertical or horizontal direction.

We consider the latter.

Without loss of generality, we assume that a matrix $A \in \mathbb{R}^{m \times n}$ with $m > n$ is traversed from the top left corner . The traversal of A is identical to the square case until the rightmost column is reached. At this point, the matrix is partitioned as follows:

$$A = \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix},$$

where $A_{TL} \in \mathbb{R}^{m \times m}$ and $A_{BL} \in \mathbb{R}^{(n-m) \times m}$; both A_{TR} and A_{BR} have a width of 0 columns.

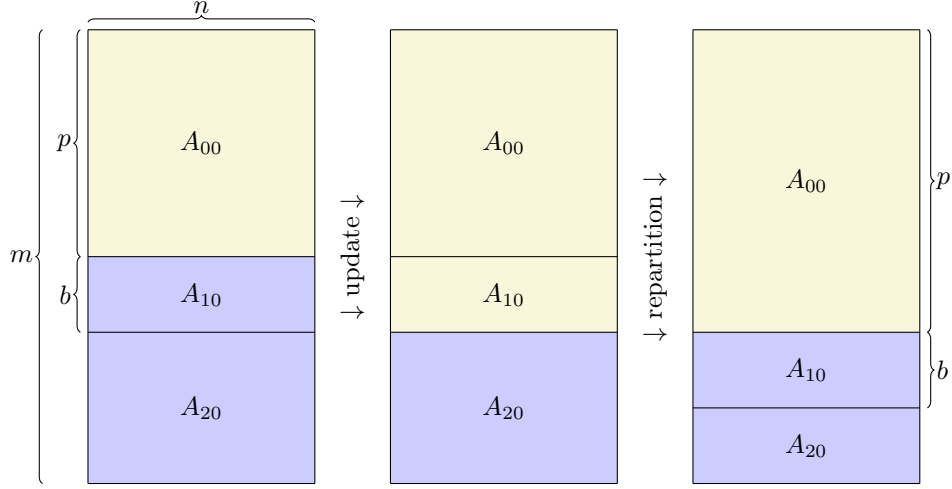


Figure 1.4: Blocked algorithms — update and repartitioning for a non-square matrix.

From this point on, the matrix is repartitioned, where all submatrices that originate from A_{TR} and A_{TL} have 0 columns:

$$A = \begin{pmatrix} A_{TL} \\ A_{BL} \end{pmatrix} = \begin{pmatrix} A_{00} \\ A_{10} \\ A_{20} \end{pmatrix}.$$

The new submatrices are of size $A_{00} \in \mathbb{R}^{p \times n}$, $A_{10} \in \mathbb{R}^{b \times n}$, and $A_{20} \in \mathbb{R}^{(m-p-b) \times n}$ (see Figure 1.4). The updates are applied to these submatrices as usual. Those that involve empty matrices do not have any effect.

At the end of the iteration, the matrix is partitioned as

$$\begin{pmatrix} A_{00} \\ A_{10} \\ A_{20} \end{pmatrix} = \begin{pmatrix} A_{TL} \\ A_{BL} \end{pmatrix} = A$$

and we update $p \leftarrow p + b$. The algorithm terminates, once $p = m$ (and inherently $p \geq n$).

Chapter 2

Sampling

In this section, we describe the construction of a tool for performance measurements of dense linear algebra (DLA) routine executions: The *Sampler*.

We start by specifying our goal and the desired functionality of the Sampler ([Section 2.1](#)). Subsequently, we conduct a set of experiments that will aid us in the design and implementation of the Sampler ([Section 2.2](#)). In [Section 2.3](#), we introduce the Sampler and discuss its design and interface.

2.1 The Goal

Our goal is to design a performance measurement tool that can be used as a basis for performance modeling. In the following, we will specify the Sampler’s desired functionality, discussing

- the performance metrics that are available and of interest to us ([Section 2.1.1](#)),
- the interface used to specify measurement requests ([Section 2.1.2](#)), and
- the configuration of the environment for the measurements ([Section 2.1.3](#)).

2.1.1 Performance Metrics

Before designing our performance measurement tool, we first need to clarify and structure our understanding of performance.

We consider the performance of a routine execution to be a set of *performance metrics*. These describe several aspects of the execution and events occurring during its runtime. We divide performance metrics into two types:

- Intrinsic performance metrics — henceforth called *performance counters* — are available through special CPU registers;
- *Derived performance metrics* are computed from performance counters and further information on the execution environment.

The time stamp counter, the most fundamental performance counter, is a register that is incremented once per CPU cycle. This register’s value can be accessed directly through the x86 instruction RDTSC (read time stamp counter). From now on, we use this highly accurate measure for execution time; the corresponding performance counter is referred to as *ticks*.

In order to measure further hardware events, each CPU offers between 2 and 5 hardware counters that can be configured to count certain types of events. We use the Performance Application Programming Interface version 4.2.1.0 (PAPI) [21] to access these event counter.

PAPI provides functions to configure, start, and read the counters. It supports up to 107 different events; usually only a subset of these are available on a particular CPU. These events include, but are not limited to:

Cache access events. For each cache level, PAPI can measure the number of cache reads, writes, hits, misses, and total accesses. Data and instruction caches can be treated separately or combined. On some CPUs, PAPI can distinguish between cache misses occurring during load and store instructions. On multicore CPUs, the occurrences of cache coherency related events such as accesses to shared cache lines or cache line invalidations can be counted through PAPI.

We denote the number of Level-1 and Level-2 cache misses by the performance counters *L1misses* and *L2misses*, respectively.

Translation Lookaside Buffer (TLB) events. PAPI provides counters for the number of TLB misses; data and instruction TLBs can again be treated separately or combined. We call the performance counter representing the total number of TLB misses *TLBmisses*.

Instruction events. PAPI can count the number of instructions of specific types, including: load, store, branch, and floating point instructions (with further subdivisions). The performance counter for the number of floating point operations is subsequently called *flops*¹. Further, the number of CPU cycles that specific instruction units are idle or stalling due to memory accesses can be measured separately.

The complete list of event counters and those which are available on a specific CPU can be accessed through the shell command `papi_avail`.

Derived performance metrics are computed from the performance counter and (possibly) additional information, for instance the CPU frequency or cache sizes. A list of common derived performance metrics is given below:

- Floating point operations per second can be computed from *flops*, *ticks*, the CPU's frequency *hz* and the CPU's available floating point instructions per cycle *fpipc*:

$$flops/s = \frac{flops \times hz}{ticks \times fpipc}.$$

- The *efficiency* of a routine execution is computed from *ticks*, *fpipc*, and the number of mathematical operations performed *mops*²:

$$efficiency = \frac{mops}{ticks \times fpipc}.$$

- Level-1 cache miss ratio is obtained from *L1misses* and the number of Level-1 cache accesses *L1accesses*:

$$L1missratio = \frac{L1accesses}{L1misses}.$$

For the sampling tool, we only consider the directly measurable performance counters.

¹ *flops* is the *number* of floating point operations. The number of floating point operations *per second* is consistently denoted by *flops/s*. The property of a certain CPU that determines the maximum number of floating point operations *available* per second is referred to as *peak_flops/s*.

² Routines often perform more *flops* than an operation requires from a mathematical point of view.

2.1.2 Routines and Arguments

We are interested in sampling dense linear algebra routines, such as BLAS or unblocked algorithms.

When sampling a routine, we need to specify both its symbol and its arguments. Let us consider the following BLAS call:

```

      side uplo transA diag  m    n  alpha  A  ldA  B  ldB
dtrsm( R , L , N , U , 512, 128, 0.37, A , 256, B , 512).

```

This invocation performs the operation $B \leftarrow 0.37BA^{-1}$, where $A \in \mathbb{R}^{128 \times 128}$ is lower triangular and has leading dimension 256 and $B \in \mathbb{R}^{512 \times 128}$ with leading dimension 512. We can execute the above call with different BLAS implementations. Thus, the routine can only be identified by both its name and its implementation.

Since all BLAS implementations use the same interface, we cannot use several implementations in the same program. The simplest possible solution is to generate separate sampling programs, each linked with a different BLAS implementations. In each of these programs, a routine is uniquely identified by its name.

All parameters apart from **A** and **B** are basic data types such as characters, integers, and floating point numbers; **A** and **B** on the other hand are matrices. At this point, we can exploit, that most dense linear algebra operations are mostly independent of their matrices' values; the floating point operations that are performed are solely determined by the other arguments. This means, all we need to sample a routine are sufficiently large memory chunks assigned to matrices and vectors such as **A** and **B**. Therefore, a request to sample the above call to **dtrsm** can be represented as the following tuple:

$$(\text{dtrsm}, R, L, N, U, 512, 128, 0.37, 256 \times 128, 256, 512 \times 128, 512).$$

Here, the data arguments have been replaced by their sizes in memory: **A** has leading dimension 256 and 128 columns. Thus, **dtrsm** accesses a range of $256 \times 128 = 32,768$ double precision floating point numbers for **A**.

2.1.3 Systems and Environment

The remaining concern is to control the sampling environment. This consists of the execution environment of the Sampler and the preconditions for each sample. The relevant aspects are the system architecture and the configuration of the BLAS library, such as support for multithreading.

From within the sampler, we can influence the data locality of routine arguments, which may greatly influence performance. We need a mechanism to control where the routine's arguments reside in memory. Experiments in [Section 2.2](#) will give us further understanding regarding the influence of memory locality on performance. This in turn will aid us in the design of our tool.

The last important requirements for the Sampler are minimal overhead and performance distortion. Both can be achieved by a clean design and structure.

2.2 Experiments

We now consider at a series of preliminary experiments that guide our design of the Sampler. In particular, we execute **dtrsm** to analyze the effects of various setups.

The system configuration remains unchanged, that is:

- One core of an Intel Harpertown E5450 processor running at 2.99GHz;
- Intel's C Compiler (**icc**).

	GotoBLAS2 (●)		MKL (●)		ATLAS (●)	
	1 st	median	1 st	median	1 st	median
<i>ticks</i>	5,737,806	497,790	12,408,516	451,026	224,973,414	1,081,404
<i>flops</i>	611,992	611,785	646,350	609,784	1,217,632	1,216,491
<i>L1misses</i>	38,762	38,428	98,996	28,690	15,248	13,925
<i>efficiency</i>	5.30%	61.1%	2.45%	67.4%	0.135%	28.1%

Table 2.1: Initial measurement outliers.

We use the high performance BLAS implementations GotoBLAS2, MKL, and ATLAS³.

The first experiment (Section 2.2.1) is discussed in greater detail and serves as a reference for modifications of the setup (Section 2.2.2).

2.2.1 Reference Experiment

In the first experiment, we repeatedly execute

```
side  uplo  transA  diag  m  n  alpha  A  ldA  B  ldB
dtrsm( R ,  L ,    N ,  U , 128, 96, 0.37, A , 128, B , 128),
```

where A and B are assigned fixed memory locations across all repetitions.

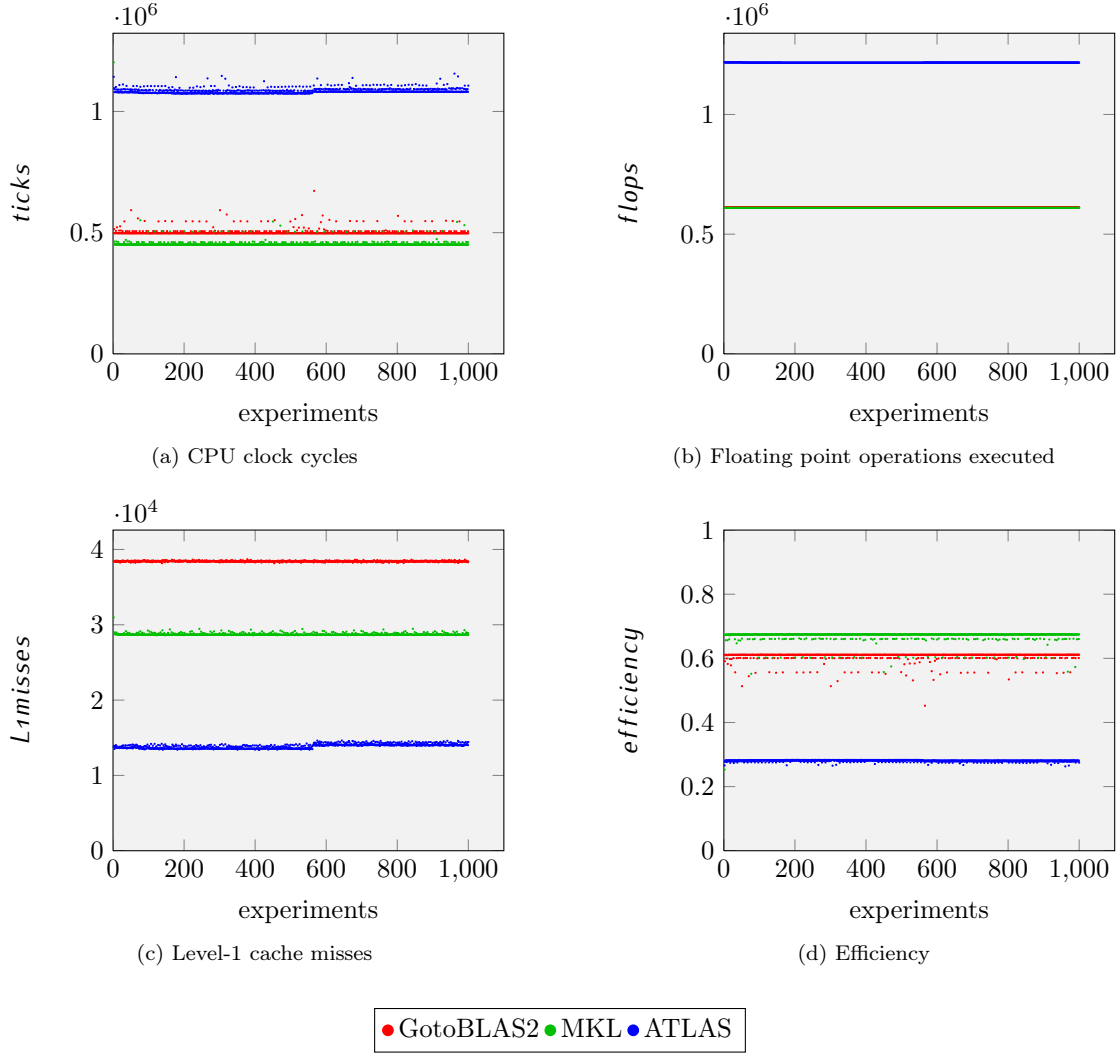
We measure the performance counters *ticks*, *flops*, and *L1misses*, and compute the derived performance metric *efficiency*. The measurements are performed as follows:

1. The performance counters are initialized using PAPI;
2. The routine is executed;
3. The performance counters are read through PAPI;
4. The results are written to a file;

These steps are repeated 1000 times, collecting all results.

The first execution of each series of experiments reproducibly yields a significantly higher and thus less efficient result. Table 2.1 shows these series' first measurements and their medians. Especially MKL's first result is very large. These outliers are due to the BLAS implementations, which, when called for the first time, perform a number of initializations, such as memory allocations, system architecture identification, and algorithm selection. From the observation of the initial outliers, we learn to discard the first experiment's result in each program invocation. We do not integrate this idea into the Sampler, since we might later use it to analyze exactly this behavior. However, we have to keep the outliers in mind when we use the Sampler to measure performance, for example, for the construction of performance models in Section 3.2.1.

All but the first outlier measurement are shown in Figure 2.1. We observe the following: For the faster implementations GotoBLAS2 (●) and MKL (●) the measured *ticks* and, thus, the computed *efficiency* fluctuate in a range of 8%. *flops* and *L1misses* on the other hand are almost constant. GotoBLAS2 (●) and MKL (●) attain efficiencies of 61% and 67%, respectively; this relatively low performance is due to the small problem size of 96×128 .

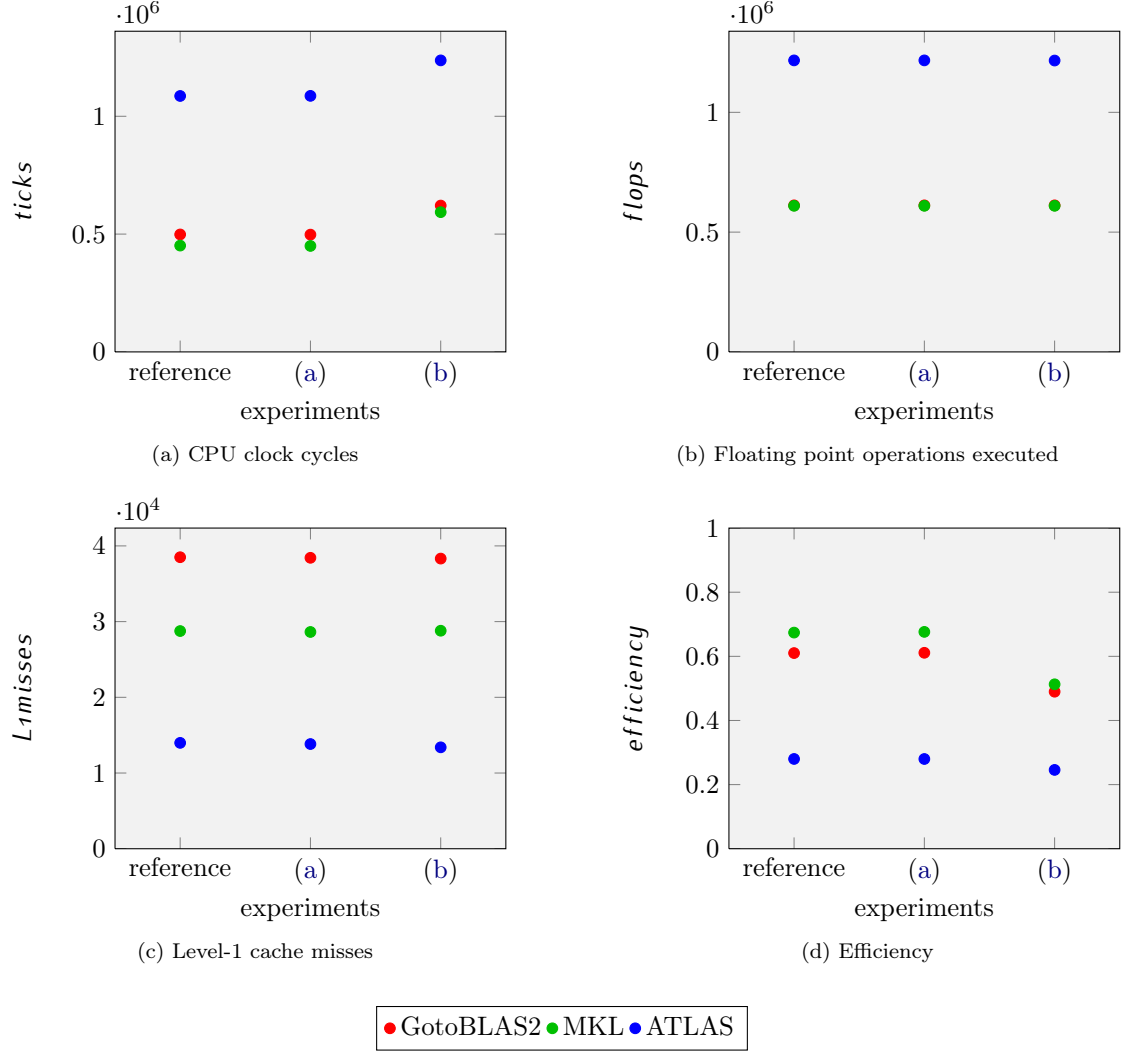
Figure 2.1: Repeated execution of `dtrsm`.

2.2.2 Modified Setups

We now modify the experiment setup used in [Section 2.2.1](#) and investigate its influence on the measurements. The results of the modifications for the same performance metrics as in the reference experiment are shown in [Figure 2.2](#). Each experiment was executed 1000 times and the corresponding plots show the series' median.

- (a) *Separation of sampling and IO*: In this modification the 1000 repeated routine invocations are executed consecutively without writing the results to the output. Instead, the results are stored in memory and only written to the standard output stream after all 1000 samples were taken. As we can see in [Figure 2.2](#), this modification did not affect the performance measurements.

³See [Section A.4](#) for further detail on these implementations.

Figure 2.2: Modifications of the experiment setup for `dtrsm`.

However, we observe a slight decrease in the total runtime of the experiment series of 0.7 seconds, or 0.7 milliseconds (2,137,850 ticks) per sample.

- (b) *Random matrix memory locations*: In this modification, the routine's matrix arguments are randomly taken from a 2GB large memory chunk. Using this configuration, the results differ significantly: While *flops* and *L1misses* remain unchanged, we experience an increase in execution time and decrease in efficiency. This is expected, since the number of floating point instructions is unchanged, while the memory accesses are to main memory instead of the CPU's caches. Both reusing the same memory locations to gain increased memory locality, as well as accessing memory areas which do not reside in the caches — called *cache trashing* — are interesting scenarios.

2.3 The Sampler

With gained insights into the relevant factors for measuring performance, we turn to the Sampler. This tool is designed to fulfill the requirements specified in [Section 2.1](#). [Section 2.3.1](#) gives an overview of the design and mechanisms used in the Sampler’s implementation. We then turn to its usage and interface in [Section 2.3.2](#). The tool is employed in a series of experiments in [Section 3.1](#).

2.3.1 Design

The Sampler is designed as a flexible lightweight performance measurement tool. It is written in C and can, hence, directly interfaces to libraries such as BLAS [\[20, 13\]](#) or LAPACK [\[10, 7\]](#).

The used libraries and the list of routines that can be sampled are incorporated into the Sampler during building process. Given a list of header files for the routines, specific object files are generated. These contain information on the routines signatures necessary to read and execute corresponding sampling requests. Linking these object files with the implementations of the routines yields the Sampler. The same object files can be linked with different routine implementations (e.g., GotoBLAS2 or MKL) to create separate Samplers for each of them. In order to sample multithreaded routines, the implementations need to be configured either before linking or through the system environment at runtime, according to the libraries configuration system.

We discuss the structure of the Sampler by looking at its program flow. The outline of the execution of the sampler is as follows:

1. Initialization.
2. While the end of input stream is not reached:
 - (a) Read a set of sampling requests and prepare their execution;
 - (b) Execute and sample the routines;
 - (c) Write the results to the output.
3. Finalization.

We will now consider each of these stages in more detail.

Initialization. The initialization of the Sampler starts by reading a specified configuration file (a full list of possible configurations is given in [Section 2.3.2.1](#)). In case PAPI is requested, the PAPI library is initialized.

Then, the Sampler sets up the input and output streams. These are used to read sampling requests and write sampling results. These streams, which by default are the programs standard IO streams, can be redirected to files.

Next, the Sampler allocates a large contiguous chunk of memory of configurable size. This memory is used for the vector and matrix arguments of the sampled routines. It is initialized with random double precision numbers between 0 and 1.

Reading sampling requests. Once the main loop is reached, the Sampler begins to read sampling requests from the input stream (their syntax is given in [Section 2.3.2.2](#)). For each request, the Sampler’s assigns memory locations within the preallocated memory chunk to sampled routine’s vector and matrix arguments. Different memory access patterns (such as cache trashing or in-cache) can be configured (see [Section 2.3.2.1](#)).

There are three conditions upon which the Sampler stops reading requests:

- The (configurable) maximum number of routine executions that are sampled in one block is reached;
- A special command "go" was read from the input stream;
- The end of the input stream is reached.

Executing sampling requests. In this step, the Sampler executes all sampling requests read in the previous step. The resulting measurements are stored in memory for each execution. IO and sampling are separated in order to reduce possible interference and overhead.

Writing the results to the output. Now, the measurement results are written to the output stream (the output format is given in [Section 2.3.2](#)). The structures used to handle the sampling requests are cleaned such that they can be reused in the next iteration of the main loop.

Finalization. After the main loop terminates, the Sampler frees the used memory.

2.3.2 Interface and Usage

The Sampler's interface is designed to be as clean and flexible as possible; the Sampler can be used either as an interactive tool in a shell or by other programs and scripts. The interface consists of three major parts:

- The configuration file ([Section 2.3.2.1](#)),
- The input stream format ([Section 2.3.2.2](#)), and
- The output stream format ([Section 2.3.2.3](#)).

2.3.2.1 Configuration File

The Sampler is invoked with only one argument — the path to a configuration file:

```
sampler <configuration file>
```

The configuration file consists of several options, each in an individual line. Lines beginning with a # are ignored. The following options are available:

- **input = *file*:** The input is to be read from *file*. By default the input stream is the program's standard input stream.
- **output = *file*:** The sampling results are to be redirected to *file*. The default is the standard output.
- **usepapi = 1|0:** Whether PAPI is to be used for performance counter sampling (1) or not (0). Without PAPI, only *ticks* is measured.
- **ncounters = *n*:** The number of PAPI performance counters to be used. This number is inherently limited by the systems CPU architecture.
- **counters[*i*] = *counter_id*:** The *i*-th PAPI performance counter is set to *counter_id* — any valid PAPI event name. On systems with PAPI, a full list of available event names is available through the shell command `papi_avail`.

- `maxcalls = n`: n sampling requests are to be executed in a block.
- `mem_size = n`: The Sampler assigns memory chunks to routine arguments from an allocation of n bytes.
- `mem_policy = n`: The memory policy is identified by a number between 0 and 3:
 - 0: **Static.** Each routine is assigned disjoint memory chunks from the start of the Sampler's memory. Since this configuration will assign the same memory locations over and over again, this configuration can be used to simulate cache locality.
 - 1: **Forward.** Each assigned memory chunk begins where the last ended, such that there is no overlap between chunks. Once the end of the Sampler's memory is used, the assignment starts over from its beginning. This configuration ensures that there are no overlaps between memory chunks in consecutive routine executions. It can therefore be used to achieve cache trashing.
 - 2: **Backward.** The memory chunks are assigned in backward order, starting from the end of the Sampler's memory. The concept of this policy is the same as for the Forward mode (1).
 - 3: **Random.** The memory chunks are taken randomly from within the Sampler's memory. If the Sampler's memory is large enough, the chances of overlapping memory regions is minimal, leading to cache trashing. If on the other hand, the memory allocation is small, overlapping memory assignments are more likely, even for chunks assigned to the same routine execution.

Varying the memory policy will allow to represent certain scenarios in the estimation of performance in the

- `mem_align = n`: Every assigned memory chunk will be aligned in memory by blocks of n bytes. This configuration may be used to achieve cache alignment, ensuring that each matrix and vector starts in a new cache line.
- `state_queries = 0|1`: When set to 1, the Sampler prints queries to the standard output, stating which routine type of arguments are expected (e.g., `int*` or `double*`).
- `show_progress = 0|1`: When set to 1, the Sampler prints its progress to standard output.
- `matlab_output = 0|1`: Activating this option leads to output in Matlab matrix notation. This is useful when Matlab is used to process the sampling results.

All options that take values 0 or 1 are 0 by default and may be omitted. The options for PAPI counters are only valid with a previous `usepapi = 1`. In this case, the number of specified performance counters has to match the number given in `ncounters = n`.

2.3.2.2 Input Stream Format

The format of the input stream agrees with the considerations in [Section 2.1.2](#). A request consists of a routine name followed by that routine's argument specifications. The only exception is the command `go`, which may be issued between sampling requests. Upon encountering this command, the Sampler immediately starts the sampling process, prints out the results, and only then continues to read its input stream. This command is needed when the sampler is used interactively or by another program in order to initiate the sampling process.

A request to sample

```

      side  uplo  transA  diag  m    n  alpha  A  ldA  B  ldB
dtrsm( R , L , N , U , 128, 96, 0.37, A , 128, B , 128),

```

for instance is submitted by the following line on the input stream:

```
dtrsm R L N U 128 96 v.37 16384 128 16384 128.
```

From the routine's signature, the Sampler know how to interpret the arguments.

- The capital characters are allowed for `char*` arguments.
- `int*` arguments take an integer value.
- For `float*` or `double*` arguments there are two possibilities:
 - Given an integer, a memory chunk of this size will be assigned to this argument.
 - Given the character `v` (for "value") followed by a floating point number, the number is passed to the routine as a single floating point number. This is useful to cover special cases such as 0, 1 or -1 , which might trigger separate routine branches.

2.3.2.3 Output Stream Format

The output format is very similar to the input format: For each sampling request, the name of the routine is printed, followed by its `char*` and `int*` arguments⁴. After the arguments, the sampling results are written to the output stream, followed by a new-line character.

The first sampling result is always the performance counter *ticks*. It is followed by PAPI's performance counter results as specified in the configuration file.

For instance, the output for the request

```
dtrsm R L N U 128 96 v0.37 16384 128 16384 128
```

with PAPI counters *flops* and *L1misses* might be:

```

name      char*      int*      ticks      flops  L1misses
dtrsm R L N U 128 96 128 128 1079973 1217382 13509

```

⁴ `float*` and `double*` arguments are omitted.

Chapter 3

Modeling

In the previous Chapter we discussed the *Sampler* — a tool for measuring the performance of DLA routines. From measurements obtained by the *Sampler*, we now want to construct analytical performance models. For this purpose, we introduce the *Modeler*, that based on the *Sampler*, generates a certain type of performance models automatically. These models form the base for our performance prediction and algorithm ranking ([Chapter 4](#)).

The *Modeler* is developed in the following design process:

- We begin by studying the dependence of performance on various arguments of DLA routines ([Section 3.1](#)). The focus is BLAS, the most fundamental routines.
- Using the gained understanding on the dependencies, we develop the structure of the performance model that the *Modeler* shall generate ([Section 3.2](#)).
- With the desired model structure in mind, we develop the *Modeler*, which generates these models automatically ([Section 3.3](#)).
- We conclude by evaluating how well the *Modeler* fulfills its objective — the reliable generation of accurate performance models ([Section 3.4](#)). In the process, we illustrate how the *Modeler* can be configured to obtain models with certain properties.

3.1 Preliminary Experiments

In the experiments in [Section 2.2](#), we have already determined a few interesting aspects of performance measurements:

1. The first measurement is always an outlier;
2. When a routine is executed repeatedly, most performance metrics produce fluctuating values;
3. The performance changes significantly depending on whether the routine's arguments are in cache or main memory.

All of these aspects are taken into account in the designs of our models in [Section 3.2](#). To avoid their influence in the experiments in this section, we take the following measures:

1. The first measurement is always discarded;
2. Each experiment is repeated 100 times and the median of this series is presented as a result;

3. Successive routine invocations use the same memory locations for their arguments (static memory policy).

We focus on the dependence of performance on the types of arguments encountered in BLAS. (An introduction to BLAS is given in [Appendix A](#).) We consider each argument type and its influence on performance individually:

- Discrete¹ arguments, such as `uplo` or `trans` ([Section 3.1.1](#)),
- Size arguments `m`, `n`, and `k` ([Section 3.1.2](#)),
- Scalar arguments, for instance `alpha` or `beta` ([Section 3.1.3](#)),
- Vector and matrix arguments, such as `A`, `B`, or `x` ([Section 3.1.4](#)), and
- Leading dimension or increment arguments like `ldA` and `incx` ([Section 3.1.5](#)).

Throughout the experiments in this section, we take measurements on one core of a Quad-Core AMD Opteron Processor 8356 [8] running at 2.30GHz. This processor can issue 2 double precision floating point instructions per cycle; therefore, its theoretical peak performance is $peak_flops/s = 2 \times 2.30 \cdot 10^9$ operations per second.

While we consider `ticks` as a representative performance counter in most experiments, the gained insights are also applicable for other performance counters, such as `flops` or `L1misses`.

3.1.1 Discrete Arguments

In order to understand the influence of discrete arguments on performance, we consider `dtrsm` ($B \leftarrow A^{-1}B$, A triangular). This routine is an ideal candidate, since it covers all types of discrete arguments that appear in BLAS: `side`, `uplo`, `transA`, and `diag`. For our experiments, we vary these arguments and fix the remaining ones:

```

      side  uplo  transA  diag  m  n  alpha  A  ldA  B  ldB
dtrsm(side, uplo, transA, diag, 256, 256, 0.5, A, 256, B, 256).

```

Measurements of the performance counter `ticks` for all possible combinations of discrete argument values are shown in [Figure 3.1](#). At a first glance, we can already see that every BLAS implementation shows a different behavior; we cannot find a consistent pattern across all of them. However, we can find several similarities common to at least some of the implementations.

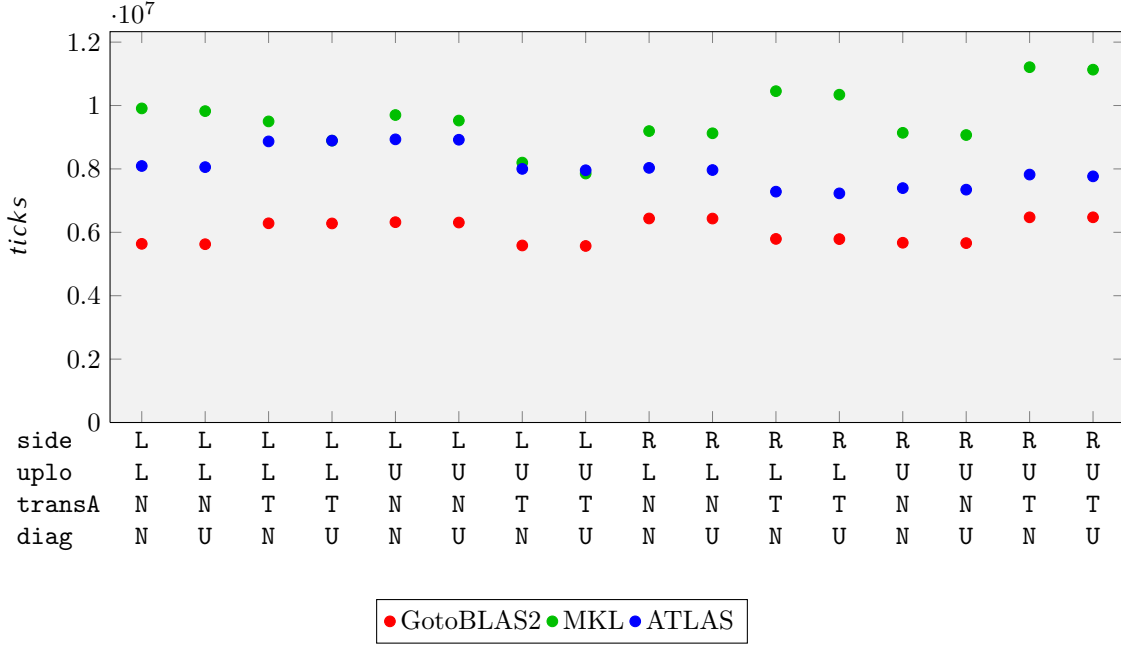
First of all, `diag` has only very little influence on performance in any of the implementations. Only for MKL² (●) and ATLAS (●), we observe that `diag = U` results in a slightly lower runtime. (This is not unexpected, since `U` indicates that A is unit-triangular, therefore requiring 256 floating point operations less.)

When we consider the discrete arguments `side`, `uplo`, and `transA`, we observe a coupled dependence. In [Figure 3.1](#), we see that the pattern on the left (`side = L`) is inverted (mirrored along a horizontal line) with respect to the same pattern on the right (`side = R`). (This is understandable, since `side` determines the order of operations — $A^{-1}B$ for `L` and BA^{-1} for `R`.) Additionally, ATLAS (●) is on average faster for `side = R`, while MKL (●) is slower for this case.

Considering `side = L` (the left half of [Figure 3.1](#)) and looking at GotoBLAS2 (●) and ATLAS (●), we find that when `uplo = L`, `transA = N` is faster than `transA = T`, while the opposite is the case for `uplo = R`; for MKL (●), `transA = T` is faster than `transA = N` independent of `uplo`.

¹ Meaning with a finite value range. Arguments that take integer values are referred to as continuous. The usage of "discrete" and "continuous" thus differs from their purely mathematical meaning.

² Since MKL is designed for Intel architectures, it attains lower performance on this AMD system.

Figure 3.1: Dependence of *ticks* on discrete arguments in `dtrsm`.

All these observations have shown that `side`, `uplo`, and `transA` have a significant impact on performance and that the dependency cannot be described in a consistent manner across all implementations. When we aim at modeling this dependency, our only option is to generate a separate model for each of combination of these parameters.

`diag`, on the other hand, was shown to only have a minor impact on performance, therefore, we are inclined to ignore this argument in order to reduce our model complexity.

3.1.2 Size Arguments

With the exception of `dgemm` ($C \leftarrow \alpha AB + \beta C$), all BLAS Level-3 routines have 2 size arguments; `dgemm` has 3: `m`, `n`, and `k`. For this reason and since `dgemm` is usually the most optimized BLAS routine, we use it in the following experiments in order to understand the dependence of performance on the size arguments. We consider the routine invocation

```

      transA  transB  m  n  k  alpha  A  ldA  B  ldB  beta  C  ldC
dgemm(  N    ,   N    , m, n, k,  0.5 , A,  m , B,  k ,  0.5 , C,  m )

```

and perform three experiments; in each, we vary one of the size arguments `m`, `n`, and `k` (and the corresponding leading dimension) between 8 and 512 while the other two are fixed to 256. Figure 3.2 shows the performance metrics *ticks* and *efficiency* for these experiments. From these plots, we can learn two important things:

- Each of the three size arguments `m` (—), `n` (.....), and `k` (---) has a different influence on the performance of the routines.

Since the number of mathematical operations for `dgemm` is $mnk + 2mn$, one might expect similar dependencies on `m` and `n` and a different one for `k`. However, the results show that

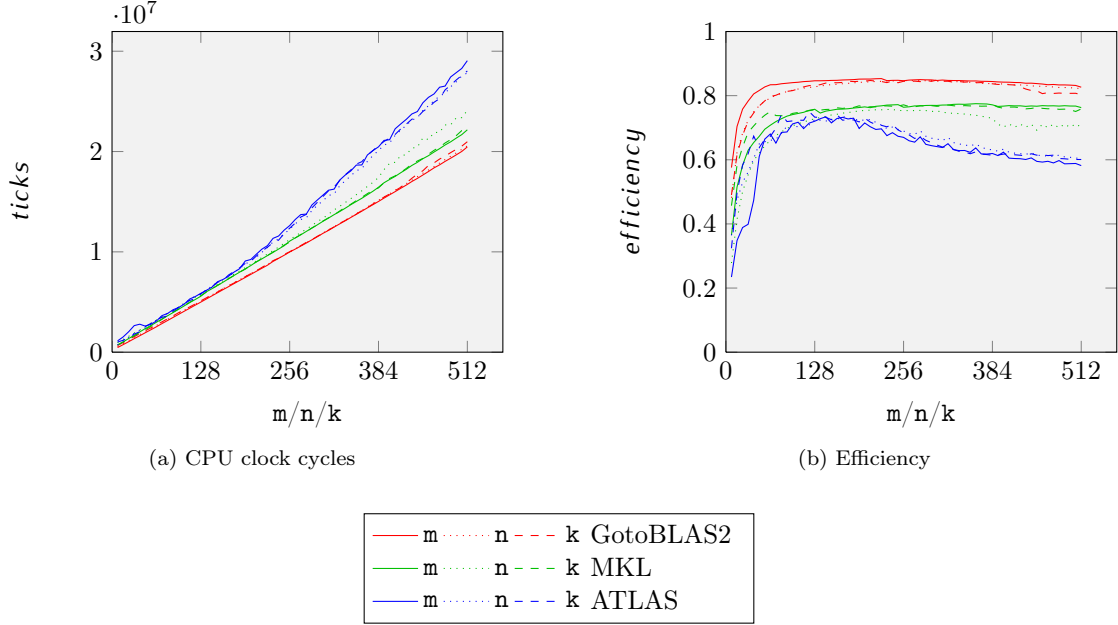


Figure 3.2: Dependence of *ticks* and *efficiency* on size arguments in *dgemm*.

this is not the case: For GotoBLAS2 (—), m is the argument which, compared to n and k , produces significantly different behavior; the same is true for MKL (—) and n .

While in this experiment, we only varied one of the size arguments while the others remained fixed, in general, all size arguments may vary independently, leading to nonlinear influences on performance. The only possibility to capture this behavior is to represent the performance as a function defined on a three dimensional argument space – one dimension for each size argument.

- The performance dependency on size parameters is generally nonlinear. Especially Figure 3.2b shows the high degree of non-linearity — in this *efficiency* graph, a linear function would result in a clean hyperbola.

However, it is possible to identify value ranges which are almost linear. For instance, we could approximate ATLAS's *ticks* (—) roughly by two linear functions: one ranging from 0 to about 200 and a second one for the remaining value range. In a similar fashion, we can approximate the performance dependency on single size arguments for GotoBLAS2 (—) and MKL (—) by several linear functions.

If we extend this approach to accommodate for all sizes arguments simultaneously, we obtain multivariate piecewise polynomials.

These experiments have analyzed the performance dependence on a fairly large scale. Now, we take a closer look at a very fine scale: We once more consider *dgemm*; this time for $m = n = k \in \{32, \dots, 64\}$. *ticks* and *efficiency* for these experiments are given in Figure 3.3.

These plots, especially for ATLAS (—), show fine oscillations. While we do not want to represent these oscillations, we have to be aware of them during the generation of models. For this purpose, we only consider samples in small intervals in order to obtain smooth performance dependencies (e.g., intervals of 8 result in —, —, and — in Figure 3.3).

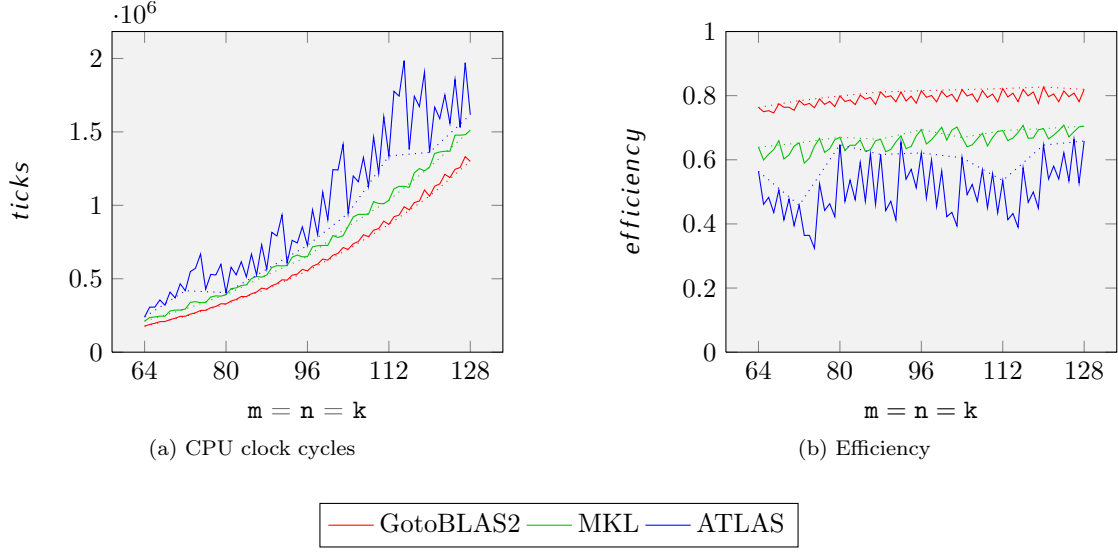


Figure 3.3: Dependence of *ticks* and *efficiency* on size arguments in `dgemm` at small scale.

3.1.3 Scalar Arguments

In BLAS, scalar arguments multiply matrices and vectors, for instance α and β in `dgemm` ($C \leftarrow \alpha AB + \beta C$). This generally leads to one multiplication (floating point operation) per entry of the scaled matrix or vector. There are a few exceptions: Multiplications by -1 , 0 , and 1 do not require multiplications. Especially -1 and 1 are very common in application codes.

In the following experiment, we once more consider `dgemm`; this time with the following arguments:

```

transA  transB  m    n    k    alpha  A  ldA  B  ldB  beta  C  ldC
dgemm(  N    ,  N    , 256, 256, 256, alpha, A, 256, B, 256, beta, C, 256).

```

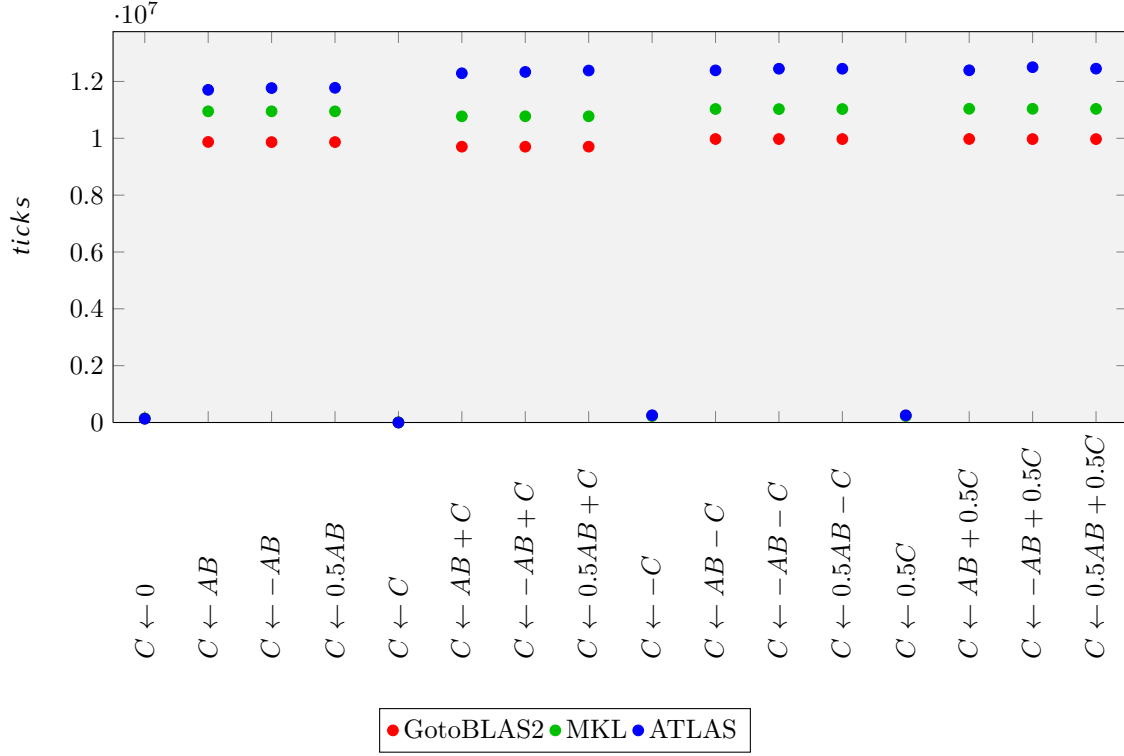
For both `alpha` and `beta` we consider the values 0 , 1 , -1 , and 0.5 (as a representative for the general case). The resulting *ticks* are shown in Figure 3.4.

Evidently, all implementations take advantage of the special cases where `alpha` = 0 : `dgemm` reduces to $C \leftarrow \beta C$ — scaling a matrix. Compared to matrix-matrix multiplication with complexity $O(n^3)$, this only requires $O(n^2)$ floating point operations. As a result the execution time drops significantly across all implementations. If now, `beta` = 1 , no work has to be done at all and the execution time is effectively 0 .

The other values for `alpha` (1 , -1 , and 0.5) do not result in different execution times.

Considering `beta`, we see that GotoBLAS2 (●) and MKL (●) show a behavior different from ATLAS (●): GotoBLAS2 (●) and MKL (●) show a decreased execution time only for `alpha` = 1 , while ATLAS (●) apparently only take advantage of `alpha` = 0 .

We can conclude that some special values of scalar arguments trigger different branches in BLAS routines, similar to discrete arguments. However, Since the exceptional case `alpha` = 0 does not appear in any well written application, the overall influence of scalar arguments on performance is rather small compared to the impact of other arguments. Therefore, we decide not to represent this dependency.

Figure 3.4: Dependence of *ticks* on scalar arguments in *dgemm*.

3.1.4 Vector and Matrix Arguments

As discussed earlier, the entries of matrix and vector arguments have no effect on the computation. Therefore, these arguments do not influence any performance metrics.

3.1.5 Leading Dimension and Increment Arguments

Leading dimension and increment arguments specify the distance of adjacent entries in a row of a matrix or vector. They can potentially influence the access patterns of both the CPU's cache and main memory, and thus affect performance.

To study the actual influence of these arguments, we once more consider *dgemm* ($C \leftarrow \alpha AB + \beta C$) and vary the three leading dimension arguments *ldA*, *ldB*, and *ldC* separately between 128 and 256. The other arguments are fixed:

```

transA  transB  m    n    k    alpha  A    ldA  B    ldB  beta
dgemm(  N    ,   N    , 128, 128, 128,  0.5 , A , ldA , B , ldB , 0.5 , C , ldC ).

```

Measurements of the resulting *ticks* and *L1misses* are given in Figure 3.5.

Again, we can observe different behaviors across our BLAS implementations: GotoBLAS2's performance (—) is independent of the leading dimension arguments, while MKL (—) and ATLAS (—) show small oscillations for increasing leading dimensions. For MKL (—), for instance, we find that *ticks* decrease slightly for increasing *ldA* and spikes again at constant intervals of 32; *L1misses* increase correspondingly.

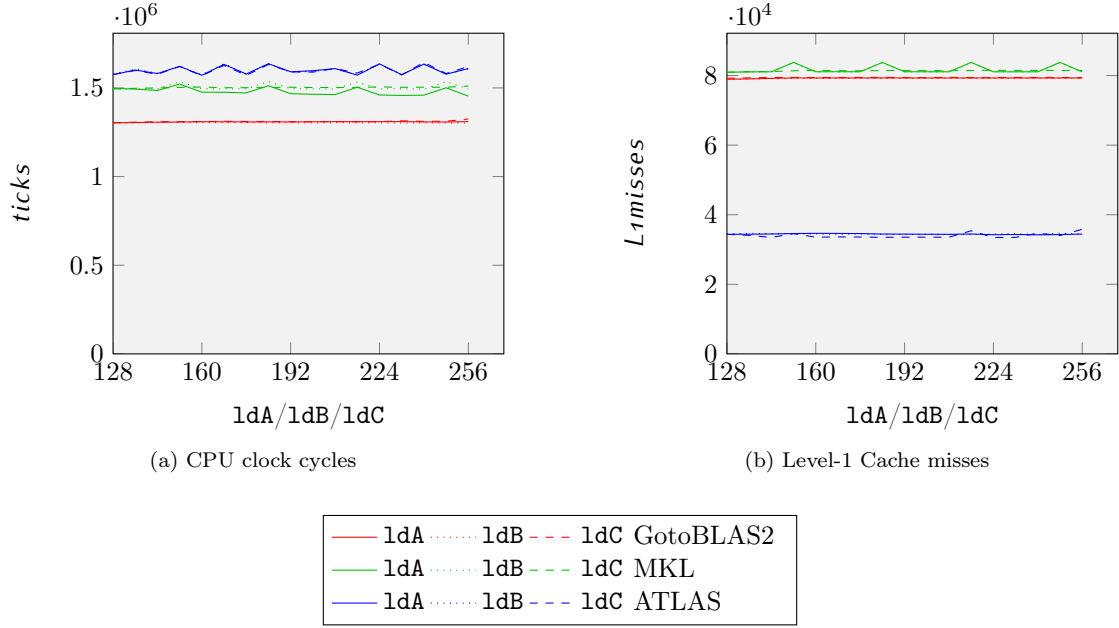


Figure 3.5: Dependence of *ticks* and *L1misses* on leading dimension arguments in `dgemm`.

Overall, however, we conclude that the leading dimension arguments only have a minor influence on performance compared to other arguments. Modeling this influence would increase the complexity of our models tremendously; we usually decide to not represent it.

3.2 The Targeted Models

Based on the observations made in the previous sections, we now introduce the type of the performance models we aim to generate. We begin by defining the model structure for the general case in [Section 3.2.1](#) and then show how a concrete model of this type may be evaluated in [Section 3.2.2](#).

3.2.1 Creation and Reasoning

In order to devise the structure for our performance models, we first recall all significant observations that we have to account for:

- Performance varies significantly between different implementations of BLAS ([Section 2.2](#) and [Appendix A.4](#)).
- Repeated executions of a single routine with fixed arguments results in fluctuating performance ([Section 2.2](#)).
- Performance is influenced by the memory locality (e.g., cache or main memory) of the matrix (and vector) arguments ([Section 2.2](#)).
- Many discrete arguments and special values for scalar arguments trigger separate branches in BLAS routines. These branches can potentially lead to a completely different performance behavior ([Sections 3.1.1](#) and [3.1.3](#)).

- (e) Performance generally depends non-linearly on size arguments are generally non-linear; however, it can often be represented by multivariate piecewise polynomials (Section 3.1.2).
- (f) Leading dimension arguments only have a minor influence on performance (Section 3.1.5).

Before we turn to the construction of models, we need to clarify what we understand under the term "performance model":

A performance model for a DLA routine is a function that, given values for the routine's arguments, provides estimates for a set of performance metrics for its execution in a certain environment.

This definition separates the observations enumerated above into two categories:

- Implementation and memory locality ((a) and (c)) depend on the execution environment. Therefore, we will create different models for different BLAS implementations and memory locality situations.
- Fluctuations and argument values ((b) and (d) through (f)) are covered by a single performance model.

To build the structure our models, we begin by considering the routine arguments. Since some of these only have very little influence on performance, we only account for a subset of arguments in our models, called the model *parameters*.

The aspects that can most significantly influence the performance behavior are discrete arguments and special values for scalar arguments (d): they can trigger the execution of different code branches. To reduce the model complexity and the effort spent on generating it, we select a subset of these — the *discrete parameters*. The argument type determines if it is modeled; **diag** and scalar arguments are usually omitted. In our models, each combination of discrete parameter values — referred to as a (*discrete*) *case* — is treated separately. Example: If we consider three discrete parameters with two possible values each, we have $2^3 = 8$ cases; each of them is modeled independently.

For every case, we introduce a separate (sub-)model for each performance metric. The remaining aspects (e), (f), and (e) are treated for each case and each performance metric separately.

We now turn to the dependence of performance on size and leading dimension arguments ((e) and (f)), which can both take non-negative integer values. In our models we offer to represent both argument types. However, we have observed that leading dimension arguments ((f)) only have a minor influence on performance. Therefore, we once more select a subset of size and leading dimension parameters (usually only the former) called *continuous parameters*³.

In principle, continuous parameters can take arbitrarily large values. We, however, limit our models to a certain range — for example, values between 8 and 1024. Additionally, in order to avoid the small scale performance oscillations (Section 3.1.2), we restrict the allowed parameter values to multiples of *mingap*: a small step size, such as 4, 8 or 16. The collection of the sets of allowed values for all continuous parameters spans a product space, referred to as (*continuous*) *parameter space*. Example: Two continuous parameters within $\{8, 16, \dots, 1024\}$ span the parameter space $\{8, 16, \dots, 1024\}^2$.

As suggested in Section 3.1.2, our choice to represent the dependencies on continuous parameters are multivariate piecewise polynomials. In 1D, these are several intervals of the parameter axis each associated with a polynomial. In higher dimension, however, piecewise polynomials could in general consist of arbitrary regions of the parameter space associated with multivariate polynomials. We

³ Continuous not in the mathematical sense but in contrast to discrete parameters, which can only take two values.

restrict the type of regions to (hyper-)cuboids with faces parallel to the parameter axes (intervals in 1D, rectangles in 2D, and cuboids in 3D). These can be represented by two points, determining lower and upper limit of the region along each coordinate direction.

It remains to incorporate aspect (b): The fluctuations of performance counters for fixed argument values. These fluctuations can only be represented in a statistical or probabilistic fashion. In principle, we would like to create a probability distribution for each performance counter at every point. Since this is not possible in practice, we limit our models to a vector of statistical quantities: minimum, average, median, standard deviation, maximum, and possibly others. Therefore, we associate each polynomial region with a vector-valued polynomial, providing these quantities. As a result, the model for one discrete case is a vector-valued multivariate piecewise polynomial.

3.2.2 Evaluation of a Model at a Point

We now show how a model of the previously introduced type is evaluated given values for the modeled routine's arguments. We take `dtrsm` ($B \leftarrow A^{-1}B$, A triangular) as an example:

```
dtrsm(side, uplo, transA, diag, m, n, alpha, A, ldA, B, ldB).
```

We consider a performance model with the following properties:

- `side`, `uplo`, and `transA` are its discrete parameters, resulting in 8 cases;
- `m` and `n` are the continuous parameters, both taken from the range $\{8, \dots, 1024\}$;
- `diag`, `alpha`, `ldA`, and `ldB` are neglected;
- It provides estimates for *ticks*, *flops*, and *L1misses*;
- It represents the performance through the statistical quantities minimum, average, standard deviation, and maximum.

Figure 3.6 visualizes the evaluation of this model for the exemplary argument values

```
side uplo transA diag m n alpha ldA ldB
dtrsm( L , U , N , N , 256, 512, 0.7 , A, 512, B, 512).
```

Given this tuple of arguments $(L, U, N, N, 256, 512, A, 512, B, 512)$, the first step in the model evaluation is to extract the model parameters. In our case, these are:

$$(\text{side}, \text{uplo}, \text{transA}, \text{m}, \text{n}) = (L, U, N, 256, 512).$$

These are separated into discrete and continuous parameters:

- The discrete parameters $(\text{side}, \text{uplo}, \text{transA}) = (L, U, N)$ determine the case;
- The continuous parameters $(\text{m}, \text{n}) = (256, 512)$ identify a point in the two dimensional continuous parameter space.

For the case (L, U, N) , the model has one vector valued multivariate piecewise polynomial $P : \{8, \dots, 1024\} \rightarrow \mathbb{N}^4$ for each performance counter; the result vector contains the statistical quantities. P is represented by a set of rectangular regions that cover the two dimensional parameter space, each associated with one vector valued piecewise polynomial. The point $(256, 512)$ lies within at least one of these regions. For the case of overlapping regions, the model accuracy of each of them is stored along with its polynomial; the region with the most accurate model is selected.

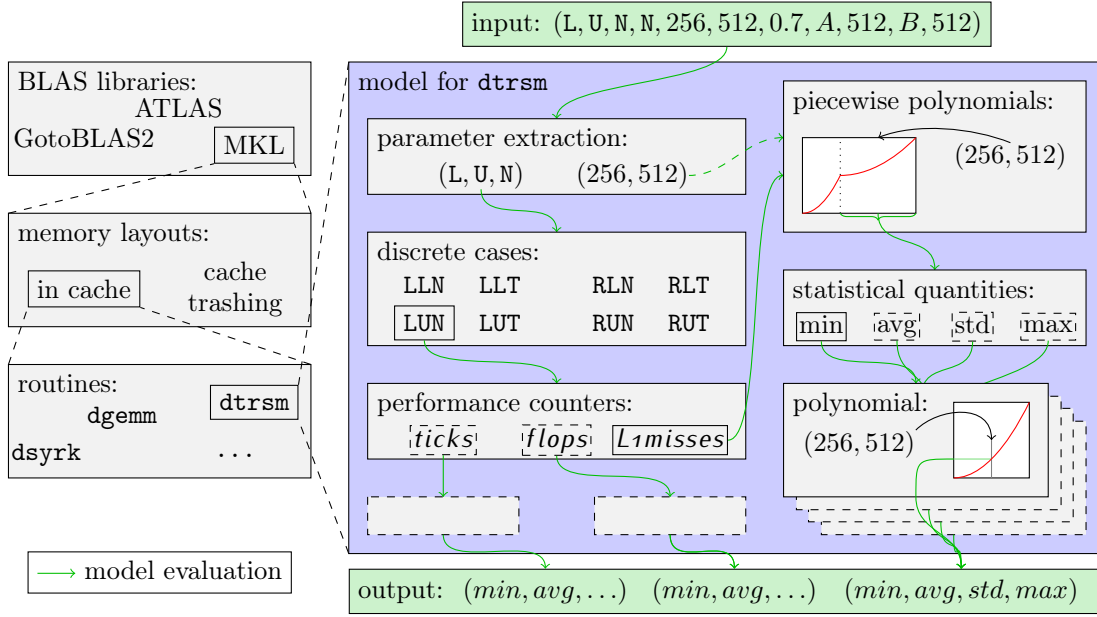


Figure 3.6: Structure and categorization of performance models and their evaluation.

For the identified region, the associated vector valued polynomial is evaluated at the point $(256, 512)$. The resulting vector contains the quantities minimum, average, standard deviation, and maximum for one performance counter.

The process — identifying the region and evaluating its polynomial — is applied to all performance counters: *ticks*, *flops*, and *L1misses*. The result of the model evaluation is a vector with the statistic quantities for each of them.

3.3 The Modeler

In the previous section, we have described the structure of our targeted performance models. We now introduce the tool that automatically generates these models: the *Modeler*.

It is implemented in Python mainly for the following reasons:

- Python has extensive high-level built-in functionality for lists and dictionaries, including functional programming constructs;
- It is completely object oriented;
- It can easily interface with other programs (in our case the Sampler);
- Its scientific library SciPy provides mathematical tools, such as least-squares solvers.

The Modeler uses an iterative approach, starting with an initial set of samples and a very rough or small models. Depending on the accuracy, more samples are taken to expand or refine the models. The main program flow is as follows.

1. Initialization:

- Read a configuration file.

- Initialize one instance of the Sampler.
 - For every routine that needs to be modeled, create a separate *Routine Modeler* (RModeler).
2. While not all RModelers have completed modeling their associated routines' performance do the following:
 - (a) Retrieve a list of sampling requests from each of the RModelers;
 - (b) Execute the sampling requests by the Sampler;
 - (c) Let the RModelers refine their models using the sampling results.
 3. Finalization: Retrieve the generated models from the RModelers and write them to a file.

The Modeler is limited to one Sampler; models for different BLAS implementations and memory locality situations can be obtained through the Modeler configuration file.

The Modeler consists of several, separately discussed components:

- The Sampler Interface ([Section 3.3.1](#));
- The RModelers ([Section 3.3.2](#));
- The *Piecewise Polynomial Modelers* (PModelers) as part of the RModelers ([Section 3.3.3](#)).

3.3.1 Sampler Interface

The Sampler Interface (SI) serves as a Python wrapper for the Sampler, handling and passing on sampling requests and returning the resulting measurements. It keeps a list of all measurements in a *memory file*; when the Modeler is executed repeatedly with the same Sampler configuration, measurements from this file are used to reduce the time spent on sampling.

Upon initialization, the SI loads the memory file corresponding to its Sampler's configuration; if no such file exists, a new memory file is created. Then, the Sampler process is started and OS pipes are used to exchange data.

The RModelers pass sampling requests to the SI in the form of tuples, for instance

```
(dtrsm, L, U, N, 256, 512, v.5, 131072, 512, 262144, 512).
```

All incoming requests are collected and processed together, once the Modeler instructs the SI to do so.

When this happens, the SI first scans its memory file for results matching the collected requests; the matches are added to an initially empty result list. Each entry in the memory file is served at most once per execution of the Modeler; Identical sampling requests receive different sampling results.

Those remaining requests that cannot be served from the memory file are passed to the Sampler. Every resulting measurement is both stored in the memory file and added to the result list. Once all sampling requests are covered by this list, each RModeler — identified by its routine's name — is passed exactly those results that correspond to its routine.

Upon termination, the Sampler Interface stores its memory file.

3.3.2 Routine Modeler

A Routine Modeler (RModeler) creates a performance model for a single routine. Its main task is to provide a layer of abstraction between two components: the Sampler Interface and the Piecewise Polynomial Modelers (PModelers), whose job is to create models of the form

$$f : \mathbb{N}^{\#\text{continuous parameters}} \rightarrow \mathbb{N}^{\#\text{statistical quantities}}.$$

The abstraction, as seen from the Modeler and the Sampler Interface, consists of the following stages:

Stage 1 Selection of parameters from the routine’s argument list;

Stage 2 Separation of discrete and continuous parameters;

Stage 3 Treatment of the discrete cases;

Stage 4 Handling of PModelers for each case and performance counter.

The RModeler provides three major functions, each of which covers the four stages of abstraction:

- Generating of sampling requests for the Sampler Interface ([Section 3.3.2.1](#));
- Passing the obtained sampling result to the PModelers in order to improve their models ([Section 3.3.2.2](#));
- Constructing and exporting the final model ([Section 3.3.2.3](#)).

3.3.2.1 Generation of Sampling Requests

The PModelers generate sampling requests in order to improve their models. These requests are processed by the RModeler according to the four stages of abstraction and then passed to the Sampler Interface.

Each PModeler constructs a piecewise polynomial model f for one performance counter p_i for one of the discrete cases c_j of the form

$$f : \mathbb{N}^{\#\text{continuous parameters}} \rightarrow \mathbb{N}^{\#\text{statistical quantities}}.$$

In every iteration of the Modeler’s main loop, it generates a list of requests l_i^j ; the requests are points within the continuous parameter space, for instance (256, 512). The RModeler needs to transform these tuples into a complete sampling requests, such as

(dtrsm, L, U, N, N, **256, 512**, v.5, 131072, 512, 262144, 512).

This is done as follows:

Stage 4 The RModeler begins by merging the request lists l_j^i of all PModelers for the same discrete case c_j into one list l_j . It is not necessary to keep track of which PModeler issued which requests, since all performance counters are measured for each sampling request. Every PModeler is afterwards given as many results to construct its model from as possible — even those which it did not request.

When l_j is created, all duplicates across *different* PModelers are removed: When two or more PModelers request samples at the same point, the maximum number of samples at this point appears in l_j — not the sum of all requests.

Next, l_j is compared with the list of sampling results for the case c_j that the RModeler has already received for previous sampling requests; it is possible that a PModeler requests samples at points that were already sampled for other PModelers. Every point in l_j that is found in the sampling results is removed from the list as many times as measurements are available for it already. This ensures that each PModeler receives at least as many samples at each point as specified. (It also means that when a PModeler wants to increase the number of samples at a point it previously requested, it needs to specify the total number of samples it requires — not the difference.)

The output of this stage of abstraction is a list of sampling points l_j for each discrete case c_j .

Stage 3 Next, the RModeler creates a list that contains all lists l_j , each associated with its case c_j : $((l_1, c_1), (l_2, c_2), \dots)$.

Stage 2 Then, the discrete parameters for each case are incorporated into the sampling requests, resulting in a single list of requests l . Requests in this list are of the form (c_i, p) , for instance $(L, U, N, 256, 512)$ when three discrete parameters constitute each case.

Stage 1 In the last step, the RModeler needs to add the routine name and values for the missing routine arguments to each request in l , for example,

`(dtrsm, L, U, N, N, 256, 512, v.5, 131072, 512, 262144, 512).`

An argument that does not correspond to a parameter is assigned a value according to its type:

Discrete arguments are given a default value.

Size arguments are also given a default value. However, usually all size arguments are model parameters.

Scalar arguments receive a default value, such as `v.5`. (It is also possible to configure these as discrete parameters, for example, with values `-1`, `0`, `1`, and `.5`.)

Leading dimension arguments are handled in one of two ways:

- They are given a large default value, such as 2500. This represents the situation where the corresponding matrices are submatrices of a larger matrix.
- They are set exactly to the height of the corresponding matrix, which is given by one of the size arguments. In order to determine the correct size argument the dependency between size and leading dimension arguments (which may in turn depend on the discrete arguments, such as `transA` or `side`) is encoded in the Python representation of the routine signature.

Increment arguments are set to either 1 or a larger default value to represent the access of a row of a matrix.

Matrix (or vector) arguments are set to the number of element that their matrices (or vectors) occupy in memory. This number is the product of the leading dimension (vector length) argument and the width of the matrix (1). Which size argument is involved may again depend on the discrete arguments and is determined by the Python representation of the routines signature.

The result of these four steps is a list of sampling requests that is passed to the Sampler Interface.

3.3.2.2 Processing Sampling Results

When an RModeler receives results from the Sampler Interface, each of them consist of the sampling request and one measurement for each performance counter.

Before the RModeler can present the results to the PModelers, they need to be processed according to the four stages of abstraction:

Stage 1 The RModeler begins by extracting the parameters from the sampling requests, which consists of values for all routine arguments.

Stage 2 These are then split into discrete and continuous parameters.

Stage 3 The results are added to the list of all results for the corresponding case, which is determined by the discrete parameters.

Stage 4 Each PModeler is then given a list of results corresponding to its case, containing only the measurements for its performance counter. The RModelers immediately use the results to improve or advance their models.

3.3.2.3 Assembly of the Model

When all models are complete, each RModeler retrieves the piecewise polynomials from its PModelers and assembles a model object. This independent object is then written to a file and later used independently of the Modeler.

The structure of this model was introduced in [Section 3.2](#) and resembles the four stages of abstraction. This becomes apparent when we consider its evaluation for given routine arguments:

Stage 1 The model parameters are extracted

Stage 2 They are separated into discrete and continuous parameters;

Stage 3 The discrete parameters determine a case;

Stage 4 For this case, all piecewise polynomials are evaluated at the continuous parameters.

3.3.3 Piecewise Polynomial Modelers

In [Section 3.1.2](#), we saw that performance depends on the size arguments of BLAS routines in the form of piecewise polynomials. Constructing such polynomials by hand is rather simple given a large number of samples. By contrast, the Piecewise Polynomial Modelers (PModelers) build models that closely fit the performance automatically and with as few samples as possible.

The process for modeling performance involves three steps. First, a set of sampling points is chosen for an initial region within the parameter space. The measurements resulting from these sampling points is modeled through least squares fitting as a polynomial. According to the polynomial's accuracy, the region is reshaped, more regions are generated or the model is accepted. These steps are repeated until the whole parameter space is covered with accurate models.

We offer two PModeler implementations, which differ in both their choice of sampling points and the strategy they use to cover the parameter space with regions. Before we present the implementations in [Sections 3.3.4](#) and [3.3.5](#), we discuss their common basis.

3.3.3.1 Polynomial Fitting through Least Squares

The approximation of a set of sampling results by polynomials through least squares fitting is a task common to all Piecewise Polynomial Modelers. Every performance counter is represented by a set of statistical quantities, such as minimum or average; each quantity is fitted by a separate polynomial. In this section, we introduce polynomial fitting for a single quantity.

The fitting procedure has the following inputs:

- A list of coordinates $\{\mathbf{x}_1, \dots, \mathbf{x}_n\} \in \mathbb{N}^d$ from the d dimensional parameter space;
- A value for each coordinate $\mathbf{v} = (v_1, \dots, v_n)^T$;
- A set of monomials⁴ $\{m_1, \dots, m_b\} \subset \{f : \mathbb{N}^d \rightarrow \mathbb{N}\}$ as a base for the polynomials.

Its goal is to find a polynomial $P(\mathbf{x}) = \sum_{i=1}^b a_i m_i(\mathbf{x})$ with coefficient vector $\mathbf{a} = (a_1, \dots, a_b)^T \in \mathbb{R}^b$ such that $\sum_{i=1}^n (P(\mathbf{x}_i) - v_i)^2$ is minimal. This is known as a *least squares problem* and can be written as $\arg \min_{\mathbf{a}} \|X\mathbf{a} - \mathbf{v}\|^2$, where

$$X = \begin{pmatrix} m_1(\mathbf{x}_1) & m_2(\mathbf{x}_1) & \cdots & m_b(\mathbf{x}_1) \\ m_1(\mathbf{x}_2) & m_2(\mathbf{x}_2) & \cdots & m_b(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ m_1(\mathbf{x}_n) & m_2(\mathbf{x}_n) & \cdots & m_b(\mathbf{x}_n) \end{pmatrix}.$$

Its solution can be obtained as $\mathbf{a} = (X^T X)^{-1} X^T \mathbf{v}$ or by the means of singular value decomposition (SVD). We use the function `linalg.lstsq()` provided by Python's SciPy package, which is based on SVD.

For small values of \mathbf{x}_i and v_i , this least squares solver yields very accurate results. If, however, the \mathbf{x}_i and v_i are concentrated in a small area far away from the origin, the solution accuracy decreases significantly. This is not a problem of the solution method but rather due to the conditioning of the least squares problem.

In order to improve the solution accuracy, we need to improve the conditioning of X . For this purpose, we translate the problem to the origin, solve the translated problem, and translate the solution back to the original problem (see Figure 3.7). We begin by translating the coordinates \mathbf{x}_i and values v_i (●) to $\tilde{\mathbf{x}}_i$ and \tilde{v}_i (●) such that they are evenly distributed around the origin:

$$\tilde{\mathbf{x}}_i = \mathbf{x}_i - \frac{1}{n} \sum_{j=1}^n \mathbf{x}_j \text{ and } \tilde{v}_i = v_i - \frac{1}{n} \sum_{j=1}^n v_j.$$

These quantities form a new least squares problem $\arg \min_{\tilde{\mathbf{a}}} \|\tilde{X}\tilde{\mathbf{a}} - \tilde{\mathbf{v}}\|^2$; its solution defines a polynomial $\tilde{P}(\mathbf{x}) = \sum_{i=1}^b \tilde{a}_i m_i(\mathbf{x})$ (—). The solution P (—) to the original problem is now obtained through a second translation:

$$P(\mathbf{x}) = \tilde{P}\left(\mathbf{x} - \frac{1}{n} \sum_{j=1}^n \mathbf{x}_j\right) + \frac{1}{n} \sum_{j=1}^n v_j.$$

⁴ A polynomial with only one non-zero term, such as 1, x_1 , or $x_1 x_2^2$.

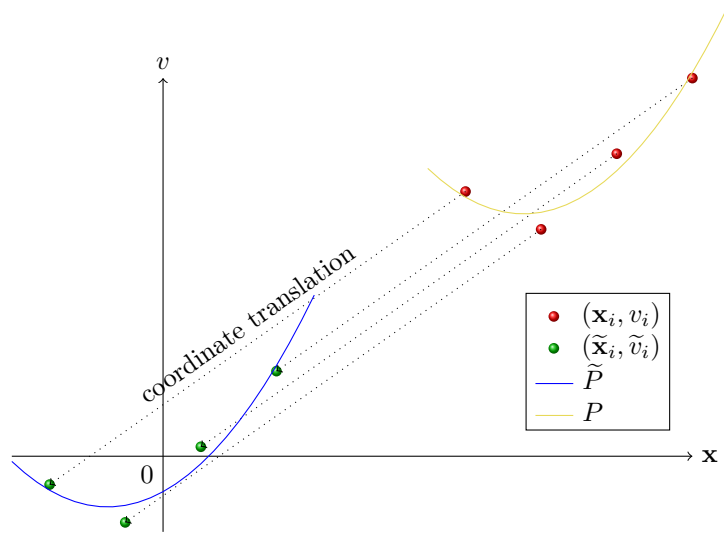


Figure 3.7: Translation of the least squares problem.

In addition to this coordinate translation, our least squares fitting mechanism rounds coefficient to close rational numbers or discards relatively small coefficient according to a configurable threshold. This is especially useful for performance counters such as *flops*, where all coefficients are usually rational numbers with small denominators.

3.3.3.2 Approximation Accuracy

The accuracy of a polynomial approximation P for coordinates \mathbf{x}_i and values v_i is determined by the local errors $e_i = P(\mathbf{x}_i) - v_i$. The used least squares approach minimizes $\sum_{i=1}^n e_i^2$; this quantity could be used to measure the approximation accuracy. We, however, use the *maximum relative error* across all \mathbf{x}_i :

$$e_{\text{relmax}} = \max_{1 \leq i \leq n} \frac{|e_i|}{v_i}.$$

For a vector valued polynomial representing the statistical quantities, we select the approximation error in one of the statistical quantities to represent the approximation's accuracy; usually, we choose the median.

3.3.4 PModeler: Model Expansion

In this section, we introduce the first of two Piecewise Polynomial Modelers (PModelers) presented in this thesis. The *Model Expansion* strategy creates piecewise polynomials as follows:

- It begins by modeling a small region in a corner of the parameter space through a single polynomial (Section 3.3.4.1);
- This region is expanded as far as possible (Section 3.3.4.2), as long as
 - its polynomial's approximation accuracy is above a given threshold, and
 - it stays within the boundaries of the parameter space;

- Once a region cannot be extended further, new adjacent regions are generated (Section 3.3.4.3).

The process of modeling, expansion, and region generation is repeated for all regions until the whole parameter space is covered.

Models are expanded either from the origin of the parameter space towards its maximum or in the opposite direction. The initial domain is accordingly chosen at the minimum (bottom left) or maximum (top right) corner of the parameter space. Without loss of generality, we introduce Model Expansion by considering the expansion away from the origin.

3.3.4.1 Initial Model

The size of the initial region is chosen with both the resulting model and the expense of sampling in mind: Smaller regions lead to a higher resolution of the model, while larger — thus, fewer — regions require less sampling. We found a length of 64 or 128 along each parameter direction to be a good compromise.

Within the initial region, the sampling points are chosen on a regular grid. For a targeted polynomial order o , this grid has least $o + 1$ points along each parameter direction; for a d -dimensional parameters-space, the grid consists of $(o+1)^d$ points. At the expense of more sampling requests, the grid density can be increased to generate smoother models.

Once a first model is created, its approximation error is computed. If this error is below the configurable threshold, the region is expanded (Section 3.3.4.2); otherwise, new adjacent regions are created immediately (Section 3.3.4.3).

3.3.4.2 Region Expansion

While the approximation error of a region is below the target threshold, the region is expanded as far as possible, as long as

- The parameter space boundary is not reached, and
- The approximation error stays above the threshold.

Expanding away from the origin, initially, only the lower limit $\mathbf{b} = (b_1, \dots, b_d)$ (its base) of the region's extent is fixed. An upper limit, denoted by $\mathbf{c} = (c_1, \dots, c_d)$, has yet to be determined. However, we can already give both a lower and an upper bound for \mathbf{c} : The lower bound $\mathbf{l} = (l_1, \dots, l_d)$ is the upper limit of the initial region's extent; the upper bound $\mathbf{u} = (u_1, \dots, u_d)$ is the maximum corner of the parameter space. During the process of model expansion, the bounds \mathbf{l} and \mathbf{u} are increased and decreased, respectively, until they converge to one point \mathbf{c} .

We denote the region of the parameter space spanned by two points $\mathbf{a} = (a_1, \dots, a_d)$ and $\mathbf{b} = (b_1, \dots, b_d)$ by

$$R_{\mathbf{a}}^{\mathbf{b}} = \{(x_1, \dots, x_d) \in \mathbb{N}^d \mid \forall i \in \{1, \dots, d\}. a_i \leq x_i \leq b_i\}.$$

At this point either the end of the parameter is reached or the approximation error is just below the threshold.

In each step of the model expansion, a set of sampling points $P \subset R_{\mathbf{b}}^{\mathbf{a}}$ is chosen. For this purpose, a parameter value p_i is selected along each parameter direction i by means of the first of the following rules that is applicable:

- When $\frac{u_i - l_i}{2} \geq \text{maxgap}$, select $p_i = u_i + \text{maxgap}$;
- In the first step, when $u_i - l_i \geq \text{maxgap}$, select $p_i = u_i$ (this reduces the number of steps in case that $c_i = u_i$);

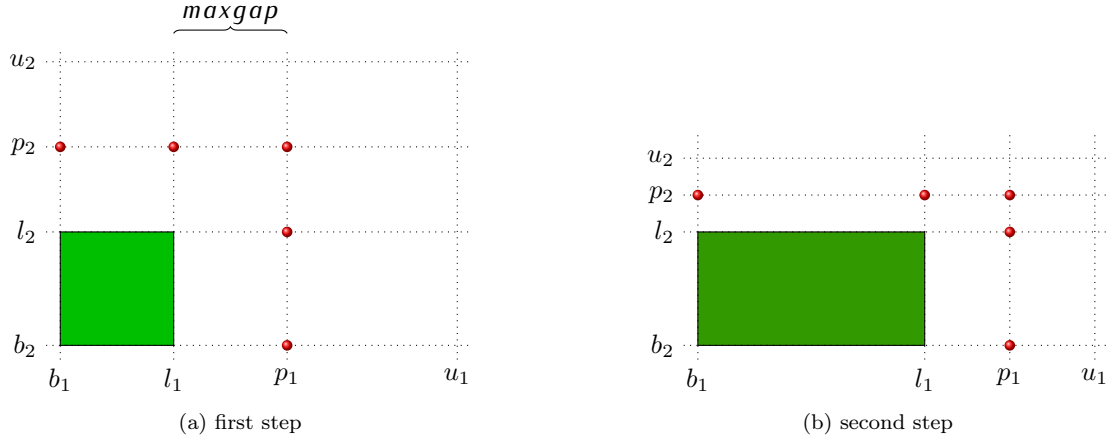


Figure 3.8: Choice of sampling points for Model Expansion.

- (c) When $l_i + \text{mingap} \geq u_i$, select $p_i = u_i$;
- (d) Otherwise, select $p_i = \lfloor \frac{l_i + u_i}{2} \rfloor_{\text{mingap}}$, resulting in a binary search pattern⁵.

This results in one sampling value p_i along each direction i .

Next, a set of sampling points P is generated from the sampling values p_i :

$$P = \left(\bigtimes_{i=1}^d \{b_i, l_i, p_i\} \right) \setminus \left(\bigtimes_{i=1}^d \{b_i, l_i\} \right).$$

Example. Let us study an example illustrating the construction of P (Figure 3.8a). We consider a two-dimensional parameter space ($d = 2$) with an initial region (■) in the lower left corner. This region determines the base $\mathbf{b} = (b_1, b_2)$ and the initial lower approximation bound $\mathbf{l} = (l_1, l_2)$ for $\mathbf{c} = (c_1, c_2)$; the upper bound $\mathbf{u} = (u_1, u_2)$ is given by the upper limit of the parameter space.

The parameter values p_1 and p_2 are determined as follows (see Figure 3.8a):

- For p_1 , rule (a) applies, since $\frac{u_1 - l_1}{2} \geq \text{maxgap}$, resulting in $p_1 = l_1 + \text{maxgap}$.
- For p_2 , none of the first three rules apply, since (b) $\frac{u_2 - l_2}{2} < \text{maxgap}$, (c) $u_2 - l_2 > \text{maxgap}$, and (c) $u_2 - l_2 > \text{mingap}$. Therefore, $p_2 = \lfloor \frac{l_2 + u_2}{2} \rfloor_{\text{mingap}}$ is chosen according to rule (d).

Given p_1 and p_2 , P (●) is constructed:

$$\begin{aligned} P &= (\{b_1, l_1, p_1\} \times \{b_2, l_2, p_2\}) \setminus (\{b_1, l_1\} \times \{b_2, l_2\}) \\ &= \{(b_1, b_2), (b_1, l_2), (b_1, p_2), (p_1, b_2), (p_1, l_2), (p_1, p_2), (l_1, b_2), (l_1, l_2), (l_1, p_2)\} \\ &\quad \setminus \{(b_1, b_2), (b_1, l_2), (l_1, b_2), (l_1, l_2)\} \\ &= \{(b_1, p_2), (p_1, b_2), (p_1, l_2), (p_1, p_2), (l_1, p_2)\}. \end{aligned}$$

The set P forms a hull around the initial region (■) with points on all intersections of b_i , l_i , and p_i . \square

⁵ $\lfloor x \rfloor_y := \max\{i \cdot y \mid i \in \mathbb{N}, i \cdot y \leq x\}$ denotes the largest multiple of y that is smaller or equal to x .

Once the sampling results for the points P are available, the expansion of the model along each direction i is considered separately. For direction i , the region of the model is tentatively expanded only along this direction up to and including p_i , resulting in a region $R_{\mathbf{b}}^{(l_1, \dots, l_{i-1}, p_i, l_{i+1}, \dots, l_d)}$. Using all available sampling results within this region — in particular, including those at p_i — a polynomial model is created through least squares fitting. Depending on how accurately the obtained polynomial approximates the sampling results, the lower or the upper bound l_i or u_i is updated:

- If the approximation error is below the threshold, region is expanded up to and including p_i by setting $l_i \leftarrow p_i$.
- Otherwise, we know that c_i must be below p_i ; therefore, u_i is set *mingap* away from p_i , since this is the largest value that c_i can still attain: $u_i \leftarrow p_i - \text{mingap}$.

When the updated bounds l_i and u_i coincide, the maximum extent of the region along direction i is reached: $c_i = l_i = u_i$. The process of sampling and modifying \mathbf{l} and \mathbf{u} is repeated until all values of \mathbf{c} are determined in the same way. Once this is the case, new regions are generated based on the current region's extent (Section 3.3.4.3).

Example. Returning to the previous in Figure 3.8a, we now consider the expansion of the region given the new sampling results. First, the region is tentatively expanded along parameter direction 1. For this purpose, the region $R_{\mathbf{b}}^{(p_1, l_2)}$ is considered; it reaches from b_1 to p_1 and b_2 to l_2 in parameter directions 1 and 2, respectively. For this region, both the samples used for the construction of the initial model and the new samples at (p_1, b_1) and (p_1, l_2) are available. From all of these samples, a new polynomial is generated through least squares fitting. Let's say that this model's approximation error is below the threshold. Consequently, l_1 is updated to: $l_1 \leftarrow p_1$.

The updated value for l_1 is now used for the tentative expansion along the second parameter direction up to p_2 . This region $R_{\mathbf{b}}^{(l_1, p_2)}$ contains both the initial model's sampling points and as all new sampling points (•). Let's say that the polynomial model created from all these points has an error that exceeds the threshold. As a consequence, the region's expansion along parameter direction 2 cannot include p_2 . Hence, u_2 is set to the largest possible value that the region's extent can attain: $u_2 \leftarrow p_2 - \text{mingap}$. The resulting situation is shown in Figure 3.8b.

Since both $l_1 < u_1$ and $l_2 < u_2$, another step of the model expansion process follows; this time the sampling points shown in Figure 3.8b are chosen. In this next step, the range of possible values for \mathbf{c} (spanned by \mathbf{l} and \mathbf{u}) is significantly smaller than before.

After only a few more steps, \mathbf{l} and \mathbf{u} will converge, yielding \mathbf{c} . □

3.3.4.3 Region Generation

Once the extent of a region has been maximized, new neighboring regions are generated. The principle behind this generation is shown in Figure 3.9: Along each expansion direction, one new region is generated right next to the maximized region. Unfortunately, it is not that simple. If we consider the two shaded regions in Figure 3.9 to be the result of the expansion process, we find that they overlap. In this case, it is undesirable to generate new regions right next to these, because they would lead to more overlapping regions and thus redundant modeling. A suitable solution for this particular case would be to generate a new region in the top right corner, bordering both regions. In this section, we introduce a procedure that generalizes this approach.

Once more considering the expansion away from the origin, the inputs of this procedure are the following:

- A region $R_{\mathbf{b}}^{c*}$ that was extended as far as possible,
- The set of all other regions \mathcal{R} , and

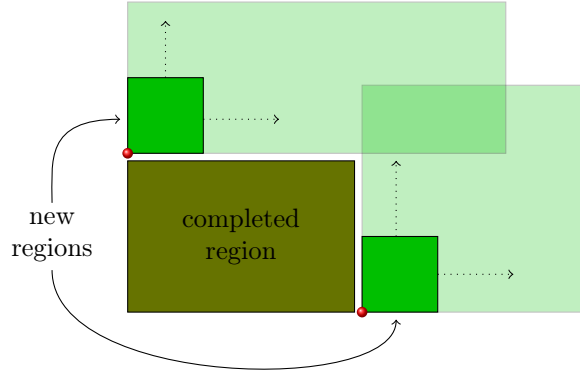


Figure 3.9: Basic concept of region generation in Model Expansion.

- The limits of the parameter space in the expansion directions.

From these inputs, the procedure generates a set of new regions. Each of these regions is identified by its base point \mathbf{b} — the lower limit of the new region's initial extent. We now describe how the set S containing these base points is created.

First, an initial set of points S is chosen; this set is then iteratively refined. The initial set consists of one point $\mathbf{p} = (b_1^*, \dots, b_{i-1}^*, c_i^* + \text{mingap}, b_{i+1}^*, \dots, b_d^*)$ for each parameter direction i . (• in the situation in Figure 3.9). Then the procedure iteratively updates S by considering the regions $R \in \mathcal{R}$ until S stays the same for all regions.

A given region R can be of two types:

- R is in the process of expansion, or
- R has been expanded as far as possible.

In the first case R 's upper limit \mathbf{c} is not fixed yet: only a lower bound \mathbf{l} and an upper bound \mathbf{u} are known, such that $\mathbf{c} \in R_{\mathbf{l}}^{\mathbf{u}}$ (see Section 3.3.4.2). With R 's base \mathbf{b} and its upper bound \mathbf{u} , its maximum possible extent is $R_{\mathbf{b}}^{\mathbf{u}}$. Every point $\mathbf{p} \in S$ that lies within this maximum extent is removed from S :

$$S \leftarrow S \setminus R_{\mathbf{b}}^{\mathbf{u}}.$$

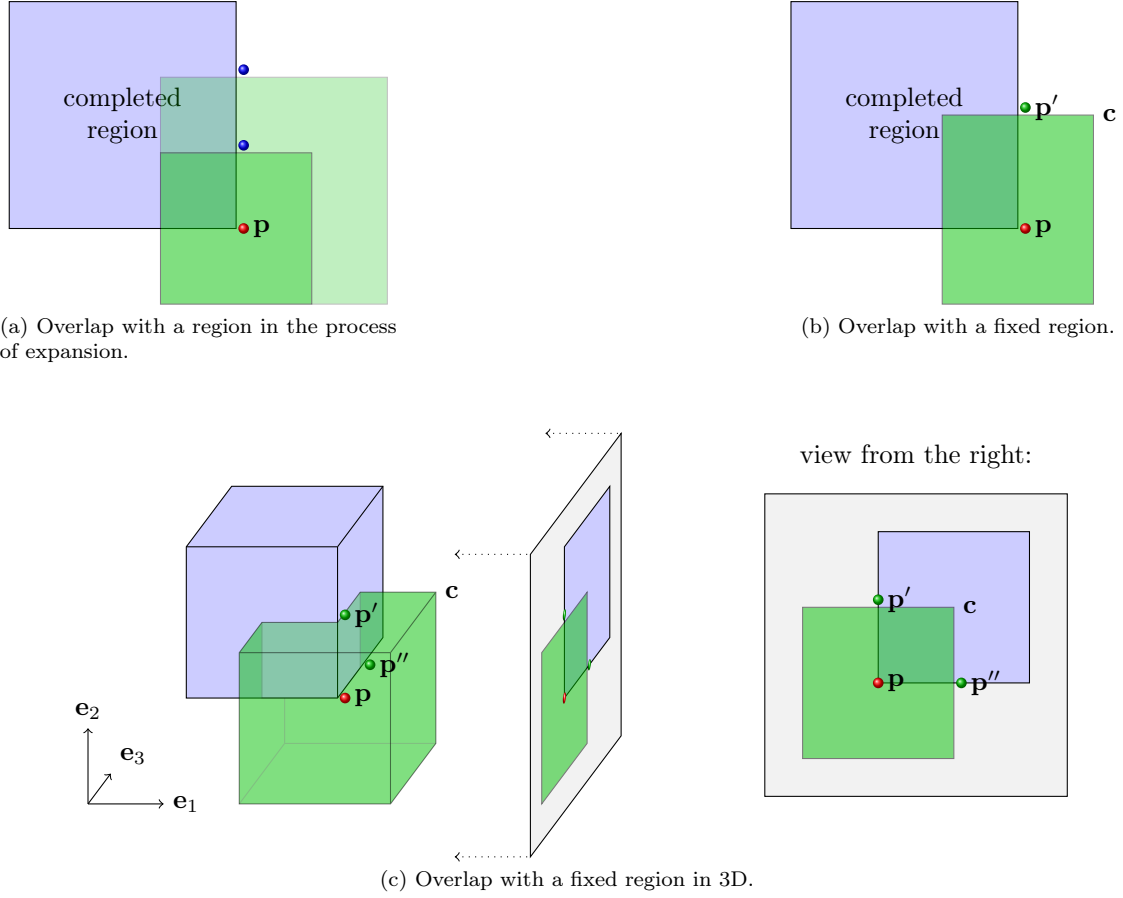
Since the extent of R is not fixed yet, \mathbf{p} cannot be selected right next to it; such a base point is created, once the expansion of R is complete.

In the case of a region $R = R_{\mathbf{b}}^{\mathbf{c}} \in \mathcal{R}$ that has been expanded as far as possible, its upper limit \mathbf{c} is known. The points $S \cap R_{\mathbf{b}}^{\mathbf{c}}$, which lie within this region are replaced by a new set of points S' . S is then updated by

$$S \leftarrow (S \cap R_{\mathbf{b}}^{\mathbf{c}}) \cup S'.$$

For this purpose, each point $\mathbf{p} \in S$ is associated with a direction p_i ; for the points in the initial set S these are the directions i , along which the base of $R_{\mathbf{b}}^{\mathbf{c}}$ was shifted to generate them. S' is created from the points $\mathbf{p} \in S$ which lie within $R_{\mathbf{b}}^{\mathbf{c}}$. For each such point \mathbf{p} , one new point is created for each coordinate direction $j \neq i_{\mathbf{p}}$ by shifting \mathbf{p} up to $c_j + \text{maxgap}$ along this direction. S' is, hence, given by

$$S' = \bigcup_{\mathbf{p} \in S \cap R_{\mathbf{b}}^{\mathbf{c}}} \{(p_1, \dots, p_{j-1}, c_j + \text{mingap}, p_{j+1}, \dots, p_d) \mid j \in \{1, \dots, d\} \setminus \{i_{\mathbf{p}}\}\}.$$

Figure 3.10: Generation of new regions' base points S .

Each point S' is associated with the directions i_p of the point \mathbf{p} from which it was generated.

The update of S through S' is the reason why all regions \mathcal{R} repeatedly until S remains unchanged: The new points within S' can lie within other regions, which were already processed. Through the repeated processing, it is guaranteed that none of the suggested new base points S lies within any other region. The termination of the process is assured since we have a finite number of regions \mathcal{R} and the new points S' are always chosen further along the expansion direction as the original point \mathbf{p} .

Example. Let us consider examples of the treatment of points in S in different situations regarding the overlap of other regions, as shown in Figure 3.10. In each example, we have a completed regions R^* (\square), an overlapping region R (\blacksquare) one a point p (\bullet), which was generated by shifting the base of the completed region R^* (\square) along direction $i_p = 1$ to the right.

- **Figure 3.10a:** In this scenario, the region R (\blacksquare) is still within the process of expansion. Creating a new region at \mathbf{p} would lead to a potentially large overlap with R . However, since the final extent of R is unknown, \mathbf{p} cannot be shifted to another position (e.g., \bullet) at the side of R . Therefore, \mathbf{p} is discarded without a replacement. Once R is expanded as far as possible, a new region will be generated somewhere between the indicated points (\bullet).

- In Figure 3.10b \mathbf{p} (●) lies within a region R (■) with a fixed limit \mathbf{c} . Therefore, \mathbf{p} is shifted along the side of R^* (□) to a point \mathbf{p}' (●), which is *mingap* away from R . With $\mathbf{p} = (p_1, p_2)$ and $\mathbf{c} = (c_1, c_2)$, we obtain $\mathbf{p}' = (p_1, c_2 + \text{mingap})$.
- Figure 3.10c shows a 3D version of the scenario from the previous example. From $\mathbf{p} = (p_1, p_2, p_3)$ (●) and $\mathbf{c} = (c_1, c_2, c_3)$, two new points $\mathbf{p}' = (p_1, c_2 + \text{mingap}, p_3)$ and $\mathbf{p}'' = (p_1, p_2, c_3 + \text{mingap})$ are generated. □

Once S is not changed anymore by any of the regions $R \in \mathcal{R}$, all points that exceed the size of the parameter space are removed from it. The resulting set gives the base points for the new regions.

3.3.5 PModeler: Adaptive Refinement

Performance depends on continuous parameters in a spatially nonuniform fashion: In some regions of the parameters space — usually for large parameter values — this dependency is rather regular and smooth, while in others it can be very irregular, containing jumps, kinks, and curvature changes. In order to represent such structures to different degrees of detail, we now introduce an approach based on adaptive refinement. Similar approaches are commonly used in a variety of disciplines such as mesh generation and optimization. In our context, the idea is to begin with a simple and regular model constructed from a coarse grid of samples; then, the model's quality and accuracy are evaluated and it is refined in the insufficiently approximated regions by locally increasing the grid resolution. These steps are applied recursively to the refined regions, until either the accuracy suffices across the whole domain or a given resolution limit is reached.

In the beginning, a single region is created spanning the whole parameter space. This region is then sampled at points on a regular grid; for polynomial approximations of order o , this grid consists of at least $o+1$ points along each direction. From the resulting samples, a first polynomial approximation is computed through least squares fitting. If the approximation error of this polynomial is below the error bound, the model is already completed. However, it is very unlikely that one polynomial is accurate enough to represent the performance of the entire parameter space.

When the accuracy of a region is insufficient, it is divided into four roughly equally sized⁶ quadrants in the 2D case (2^d hyper-cuboids in d dimensions). Those quadrants, whose size along any coordinate direction is smaller than a configurable minimum size are discarded; the others form new regions. The new regions are then sampled on a regular grid with the same number of points as for the initial region. New polynomials are fitted to the regions samples and their approximation error is computed. When this error is above the threshold, the regions are further refined recursively; otherwise, the model for the considered region is complete.

In each step of the refinement, all sampling points of the coarse model are incorporated in the model of the refined regions. In fact, the sampling grid on the refined regions covers all previous sampling points within the same region (see Figure 3.11 for the 2D case); sampling for these points is not repeated.

Example. An example of Adaptive Refinement for the 2D case is shown in Figure 3.12. The polynomial approximation for the initial region spanning the entire parameter space is very inaccurate (■, 1st square). Therefore, it is refined, generating four new regions; new samples are generated for these, leading to new polynomial approximations (2nd square). Here, the error in the top right quadrant (■) is already below the threshold (■); The other quadrants are not accurate enough and are further refined (3rd square). Now, several regions are below the error threshold; the others

⁶ The splitting point's coordinates are a multiple of *mingap* in order to avoid the small scale oscillations of performance counters.

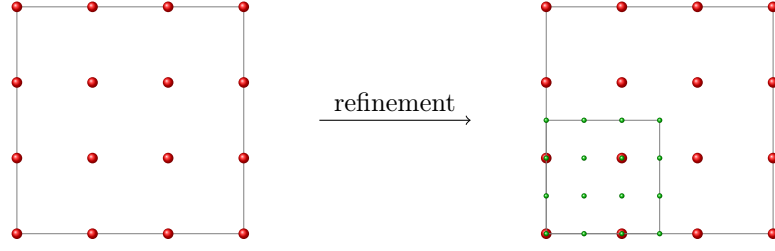


Figure 3.11: Sampling points distributed on a rectangular grid.

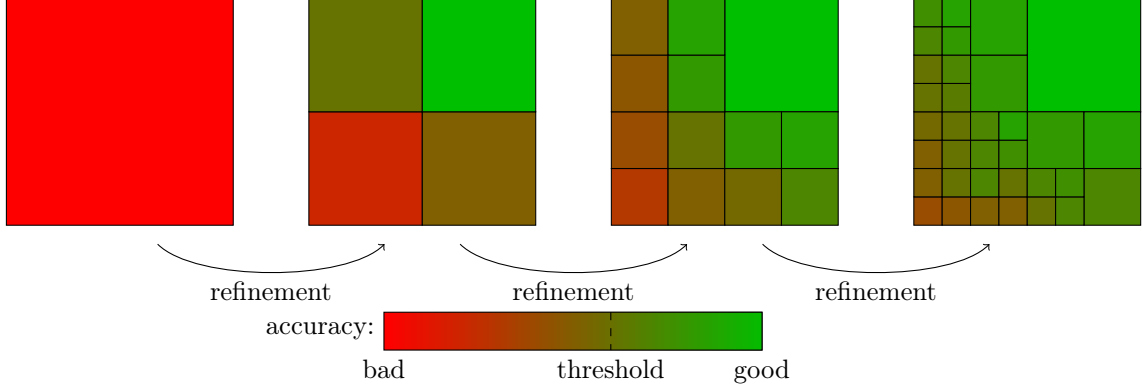


Figure 3.12: Adaptive Refinement example.

are refined once more (4^{th} square). Although some of the resulting regions are still of insufficient accuracy, they are not further refined since we do not wish to generate any smaller models. \square

3.4 Results

In the previous sections, we introduced the type of performance models we are interested in and the Modeler — a tool that automatically generates these models. We now study how the Modeler is used and configured to generate the models.

We consider `dtrsm` ($B \leftarrow A^{-1}B$, A triangular):

```
dtrsm(side, uplo, transA, diag, side, m, n, alpha, A, ldA, B, ldB).
```

This BLAS Level-3 routine has four discrete arguments (`side` through `diag`), two size arguments (`m` and `n`), one scalar argument (`alpha`) and operates on two matrices (`A` and `B` with corresponding leading dimensions `ldA` and `ldB`). From these arguments, we select

- the discrete parameters `side`, `uplo`, and `transA`, and
- the continuous parameters `m` and `n`.

The size parameters are modeled for values between 8 and 1024 with `mingap` = 8, that is, `m, n` $\in \{8, 16, 24, \dots, 1024\}$. The remaining arguments `diag`, `alpha`, `ldA`, and `ldB` are not modeled. They receive the following values:

- `diag` = `N`;

- `alpha` = 0.5;
- `ldA` = `ldB` = 2500 (representing the access of submatrices `A` and `B`).

We model the performance counters `flops` and `ticks`.

All samples for the modeling process are taken on one core of a Quad-Core AMD Opteron Processor 8356 [8] running at 2.30GHz. We use GotoBLAS2 [3, 18] and the static memory policy for high memory locality (in-cache).

Although both performance counters are modeled simultaneously, we discuss the resulting models separately in the following sections.

3.4.1 *flops*

For the performance counter *flops* (number of floating point operations performed by a routine) we can expect a constant value across routine executions with the same arguments, since the `dtrsm`'s instructions are data independent. Therefore, each sampling point is sampled only once; instead of a range of statistical quantities, we only need to consider one value at each point.

3.4.1.1 Model Expansion.

The models generated by Model Expansion are shown in Figure 3.13. The discrete parameters `uplo` and `transA` show no influence on *flops*; therefore we only consider the cases `side` = L and `side` = R. For both, we show the generated regions and their polynomials. The approximation error is 0% across the whole domain — our model represents *flops* exactly.

Considering the distribution of regions, in the case of `side` = L (Figures 3.13a and 3.13b), we have regions in intervals of 224 along the dimension `m`. Each of them is associated with a polynomial model of the form $0.5\mathbf{m}\mathbf{m}\mathbf{n} + \alpha\mathbf{m}\mathbf{n} + \beta\mathbf{n}$. Starting at 1.5, α increases by 1 in every interval, while β can be expressed as $\sum_{i=1}^n 224i$ in the n -th region. The observed regions are due to GotoBLAS2's blocked matrix access.

Considering `side` = R (Figures 3.13c and 3.13d), we find the same structure as in the previous case along the `n`-axis. This is the case, since `side` change the order of the operands from $B \leftarrow 0.5A^{-1}B$ to $B \leftarrow 0.5BA^{-1}$, thus changing the blocked data access pattern.

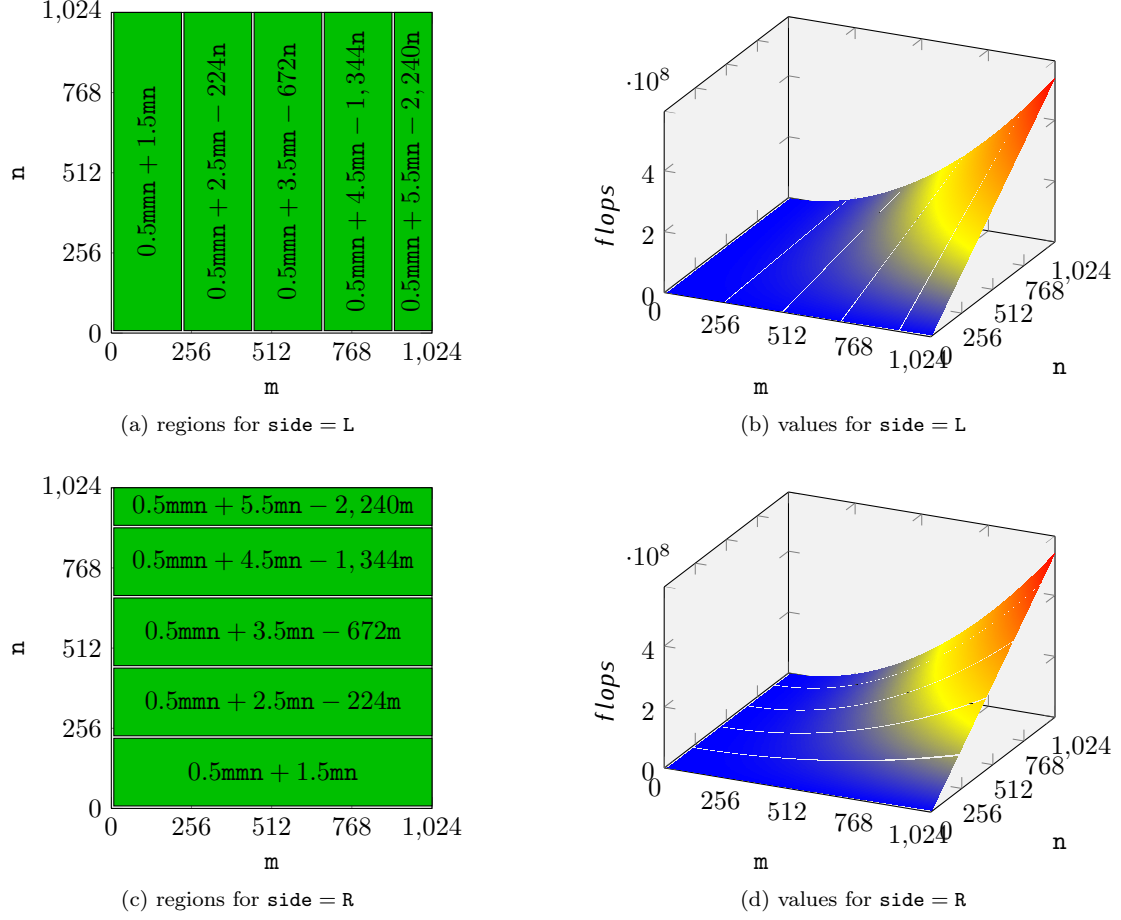
3.4.1.2 Adaptive Refinement

For `side` = L, Adaptive Refinement covers the parameter space with the regions shown in Figure 3.14. As for Model Expansion, these regions are arranged in a clear pattern, representing *ticks* perfectly. However, there are many rather small regions. This is due to the strict 2×2 subdivision pattern that is used; a comparison with Model Expansion Figure 3.13a shows that the small regions are required to exactly fill up the regions caused by GotoBLAS2's blocked matrix access.

Since we do not gain any accuracy through the small regions, we conclude that Model Expansion is more suitable to model *flops*. The accuracy of moth modeling strategies is perfect, but Adaptive Refinement requires a lot more samples than Model Expansion and does not represent the structures found in *ticks* in a clean way.

3.4.2 *ticks*

Turning to *ticks*, we are faced with a completely different performance behavior. *ticks* shows both a not clearly polynomial dependence on the continuous parameters and fluctuations when

Figure 3.13: *flops* model for *dtrsm* with Model Expansion.

one routine execution is sampled repeatedly. To accommodate for the latter, we make use of the statistical facilities of the Modeler: 10 samples are taken at each sampling point used to compute the statistical minimum, average, standard deviation, and median. We focus our analysis on the median, which we consider the most representative statistical quantity.

We focus on the discrete case $(\text{side}, \text{uplo}, \text{transA}) = (L, L, N)$. While the performance dependencies of the other cases are slightly different, they show the same structures and general behavior. Therefore, comparing different cases does not yield any insights regarding the modeling quality.

3.4.2.1 Model Expansion

Model Expansion has a few interesting configuration options for *ticks*

- The relative error bound,
- The direction of expansion (away from the origin or towards it), and
- The initial size of regions.

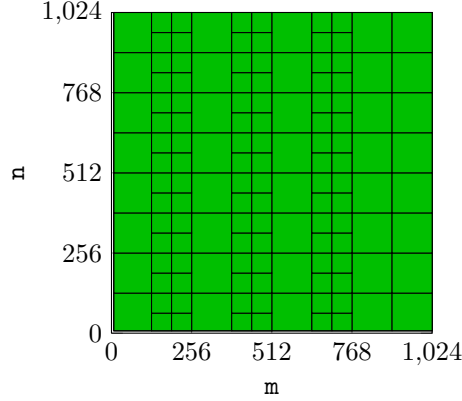


Figure 3.14: *flops* model for `dtrsm` with Adaptive Refinement — Regions for `side = L`.

We address these options' influence by considering a series of models generated with different configurations. The plots in Figure 3.15 show how these models cover the parameter space with regions along with their relative errors.

In Figure 3.15a, we begin with the following configuration:

- The error bound is 10%;
- The direction of expansion is away from the origin (\nearrow);
- New regions are initially of size 128×128 .

We observe that smaller and less accurately modeled regions are generated towards the left side of the parameter space. Towards the top right corner, the regions become larger and the relative error decreases. In this part of the parameter space, we also find several areas which are modeled by two or more overlapping regions⁷.

For the second Modeler configuration, we flip the direction of model expansion (Figure 3.15b). We observe the following changes:

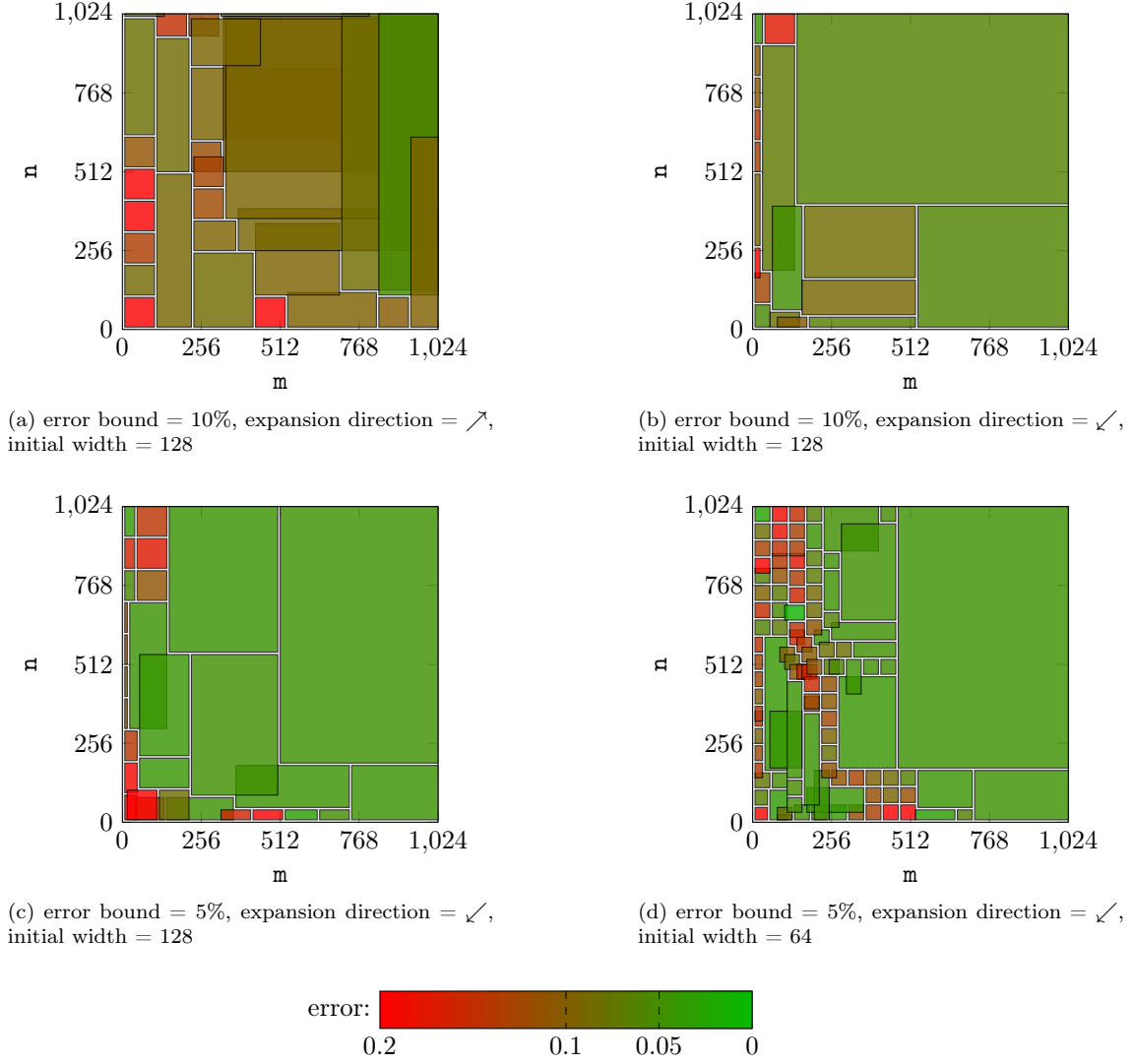
- Especially towards the top right, the generated regions are larger;
- These regions are of higher accuracy, although the error bound was not modified;
- Fewer regions overlap.

The average relative error improves from 10.4% to 6.99% while the number of required sampling points decreases from 65,220 to 32,680. We conclude that it is preferable to expand the models towards the origin (\swarrow).

In Figure 3.15c we now reduce the error bound to 5%. As a result, the average model error improves from 6.98% to 5.79%. This comes at a cost of an increase in the number of samples from 32,580 to 53,550. As in the previous cases, the accuracy of the models decreases as the parameter values become smaller; the least accurate models appear for small values of m .

Finally, we now decrease the size of initial models from 128×128 to 64×64 (Figure 3.15d). While this has little influence in the upper right part of the parameter space, lots of small less accurately modeled regions are generated for smaller parameter values. This reveals a band of inaccurately

⁷ When the model is evaluated at a point covered by multiple regions, the most accurate model is selected.

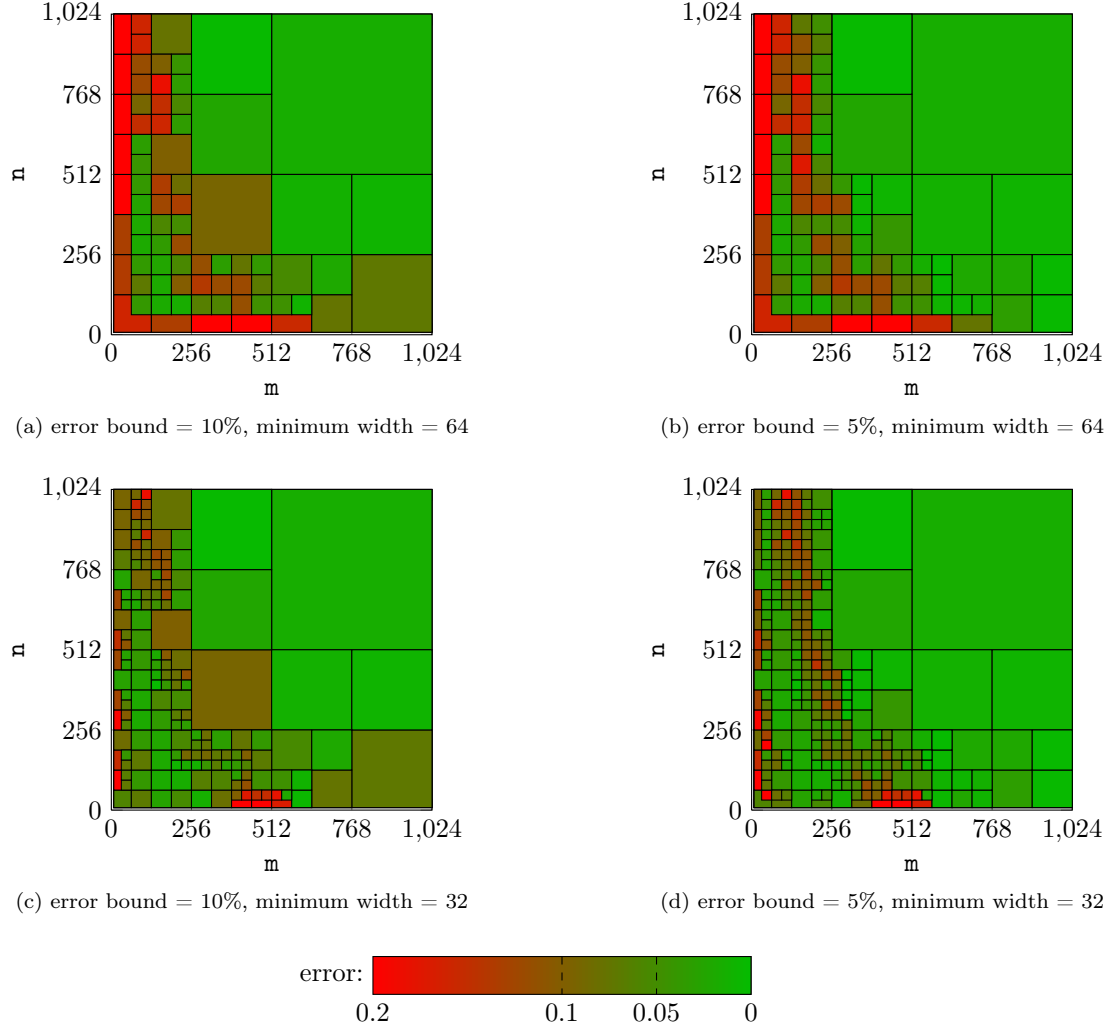
Figure 3.15: *ticks* model for *dtrsm* with Model Expansion.

modeled regions stretching from the top left corner of the parameter space ($(m, n) \approx (8, 1024)$) to the middle of its lower bound ($(m, n) \approx (512, 8)$); these artifacts were previously not visible — the sampling was too coarse. The finer sampling and the smaller models come at the cost of 84,710 additional samples — making a total of 138,290 samples. Due to the now revealed behavior of *ticks*, the average error increases from 5.79% to 5.96%.

3.4.2.2 Adaptive Refinement

The generation and distribution of regions in Adaptive Refinement is governed by two configuration options:

- The relative error bound that decides upon whether a certain region is further refined, and

Figure 3.16: *ticks* model for *dtrsm* with Adaptive Refinement.

- The minimum region size which determines the maximum depth of recursive refinements.

We now study the influence of these options individually. The models' regions resulting from varying configurations are shown in Figure 3.16.

For all configurations, the minimum region size does not allow to generate refined regions at the lower end of the parameter space. This is because the minimum value for *m* and *n* is 8 and not 0. The seemingly rectangular regions are parts of larger regions that were, for this reason, only partially refined.

The first model in Figure 3.16a was generated with an error bound of 10% and a minimum region size of 64×64 . The result shows an overall distribution of regions similar to Model Expansion: smaller and less accurately modeled regions are predominant for smaller parameter values — especially for *m*. Interestingly, we can already see a structured band of less accurate models, which was only visible in the most accurate model generated by Model Expansion (Figure 3.15d).

	error bound	expansion direction	initial width	#samples	average error
(a)	10%	↗	128	65,220	10.4%
(b)	10%	↘	128	32,680	6.98%
(c)	5%	↘	128	53,550	5.79%
(d)	5%	↘	64	138,290	5.96%

(a) Model Expansion

	error bound	minimum width	#samples	average error
(a)	10%	64	61,880	7.21%
(b)	5%	64	81,890	6.32%
(c)	10%	32	134,160	4.29%
(d)	5%	32	227,820	3.17%

(b) Adaptive Refinement

Table 3.1: Comparison of Model Expansion and Adaptive Refinement.

For our next model, we decrease the error bound to 5%, resulting in the regions shown in Figure 3.16b. To attain the higher accuracy, several of the regions in the previous model are further refined, especially on the left side. The increased number of regions is covered by 81,890 samples — 20,010 more than previously. The higher accuracy requirements lead to a decrease in the average error from 7.21% to 6.32%.

For both values of the error bound (10% and 5%), we now decrease the minimum region size to 32×32 (Figures 3.16c and 3.16d). This leads to the generation of very many tiny regions. The error bound of 10% (5%) leads to an average error of 4.29% (3.17%) at the cost of 134,160 (227,820) samples.

When we modify the accuracy and region size options, as shown in these results, Adaptive Refinement has an attractive property: All samples that were taken for the less accurate models can be reused in the generation of the finer models (made possible by the memory file of the Modeler’s Sampler Interface). A user of the Modeler can take advantage of this property and dynamically improve the models if he considers their accuracy to be insufficient.

3.4.2.3 Comparison

Table 3.1 reports the number of needed samples and the attained average error for the *ticks* models presented in the previous sections. Figure 3.17 visually compares the number of samples and accuracy of these models; we are interested in models that are as accurate as possible from a minimal number of samples.

For fewer samples, Model Expansion generates more accurate models ((b) and (d)). However, we have to keep in mind that these models were not representing the fine scale behavior of *ticks* very well. When we are willing to use more samples, Adaptive Refinement generates more accurate models ((c)). With huge amount of samples, this PModeler can generate very accurate models ((d)).

For the models used to predict the performance of blocked algorithms, we choose Adaptive Refinement with configuration (c): 10% error bound and 32×32 minimum regions size. This

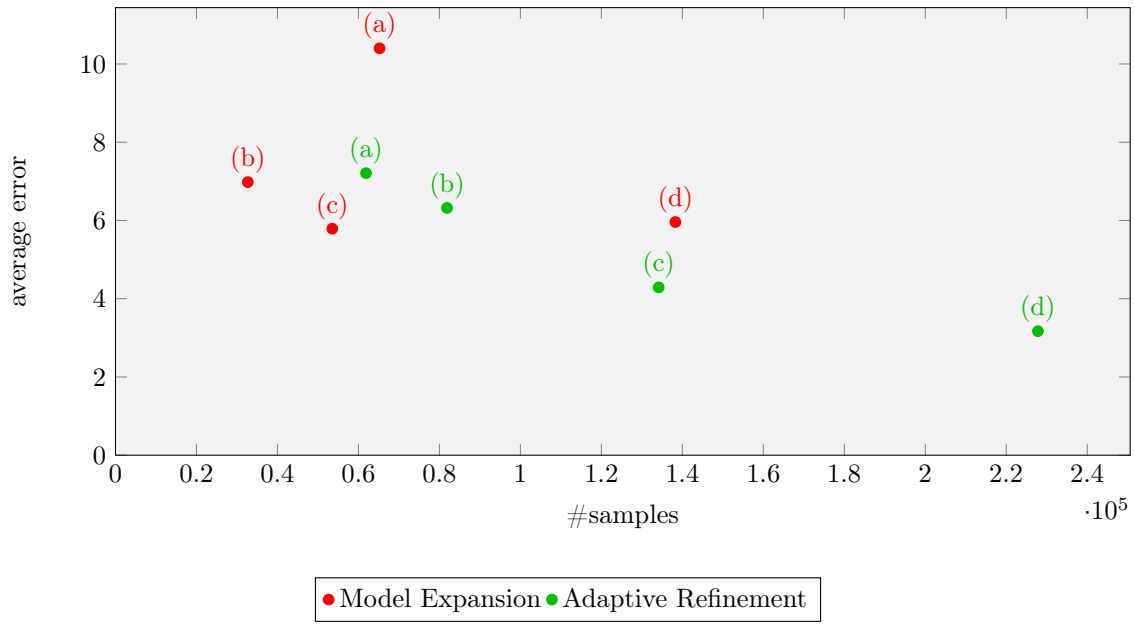


Figure 3.17: Comparison of modeling methods for *ticks*.

configuration is a good compromise between the model accuracy and the number of samples.

Chapter 4

Prediction and Ranking

We now have all necessary tools ready to tackle our main goal: ranking blocked algorithms based on performance predictions. To predict the performance of a blocked algorithm, we analyze its sequence of subroutine invocations. We use performance models generated by the Modeler to estimate the performance counters for these invocations. These estimates are then accumulated, resulting in the prediction of the algorithm's performance. The probabilistic nature of the performance model allows us to give detailed information on the expected ranges of the algorithm's performance.


Like the Modeler, our prediction tool is written in Python. The automatically generated performance models can therefore be used directly. The structure of the blocked algorithms, on the other hand, needs to be represented in a format that we can interpret within Python.

After we describe the blocked algorithm representation in [Section 4.1](#), we apply our performance prediction and ranking to three operations:

- Inversion of a triangular matrix, $L \leftarrow L^{-1}$ ([Section 4.2](#));
- LU decomposition, $LU \leftarrow A$ ([Section 4.3](#));
- Solution of the Sylvester Equation $LX + XU = C$ for X ([Section 4.4](#)).

4.1 Representation of Blocked Algorithms

To predict the performance of a certain blocked algorithm for a given set of arguments, we need a list of all subroutine invocations. In this list, each invocation is represented by a tuple consisting of the invoked routine's name and its arguments. To obtain performance estimates, these tuples can be passed directly to the performance models. The list of invocations is generated by a Python function that mimics the execution of the blocked algorithm.

Example. We consider the blocked algorithm variant 1 for the inversion of a triangular matrix $L \leftarrow L^{-1}$ ([Section 1.4.1](#)). The algorithm traverses L from the top left corner  and performs the following update statements:

Variant 1
$L_{10} \leftarrow L_{10}L_{00}$
$L_{10} \leftarrow -L_{11}^{-1}L_{10}$
$L_{11} \leftarrow L_{11}^{-1}$

```

1 int trinv1_(char* diag, int* n, double* A, int* ldA, int* blocksize) {
2     if (*n == 1) {
3         if (diag[0] == 'N')
4             *A = 1 / *A;
5         return 0;
6     }
7
8     int ione = 1;
9     double one = 1;
10    double minusone = -1;
11
12    int p = 0;
13    while (p < *n) {
14        int b = *blocksize;
15        if (p + b > *n)
16            b = *n - p;
17
18        #define A00 (A)
19        #define A10 (A + p)
20        #define A11 (A + *ldA * p + p)
21
22        dtrmm_("R", "L", "N", diag, &b, &p, &one, A00, ldA, A10, ldA);
23        dtrsm_("L", "L", "N", diag, &b, &p, &minusone, A11, ldA, A10, ldA);
24        trinv1_(diag, &b, A11, ldA, &ione);
25
26        p += b;
27    }
28    return 0;
29 }

```

Listing 4.1: Inversion of a triangular matrix — variant 1.

An implementation of this algorithm in C is given in [Listing 4.1](#). The update $L_{11} \leftarrow L_{11}^{-1}$ is mapped to a recursive routine invocation with block-size 1. This leads to the following recursion levels:

- The main algorithm invocation performs BLAS Level-3 operations on submatrices.
- At the first level of recursion, the argument `blocksize = 1` leads to matrix-vector operation. These are performed by the same BLAS Level-3 operations with one of the size parameters equal to 1.
- on the second recursion level, we have $n = 1$ — the input matrix **A** consists of a single scalar. The inversion of this scalar is handled at the very top of the routine: When `diag = N`, its reciprocal is computed.

When the algorithm is executed with the arguments

```

diag  n  A  ldA  blocksize
trinv1( N , 300, A , 300, 100 ),

```

that is, for a matrix of size 300×300 and a block-size of 100, we obtain the routine invocations listed in [Table 4.1](#). The matrix arguments do not need to be specified: Their size can be computed from the size and leading dimension arguments and the matrix entries are irrelevant to performance. Within Python, the invocation list is represented as follows:

update	subroutine invocation
$L_{10} \leftarrow L_{10}L_{00}$	(dtrmm, R, L, N, N, 100, 0, v1, ., 300, ., 300)
$L_{10} \leftarrow -L_{11}^{-1}L_{10}$	(dtrsm, L, L, N, N, 100, 0, v-1, ., 300, ., 300)
$L_{11} \leftarrow L_{11}^{-1}$	(trinv1, N, 100, ., 300, 1)
$L_{10} \leftarrow L_{10}L_{00}$	(dtrmm, R, L, N, N, 100, 100, v1, ., 300, ., 300)
$L_{10} \leftarrow -L_{11}^{-1}L_{10}$	(dtrsm, L, L, N, N, 100, 100, v-1, ., 300, ., 300)
$L_{11} \leftarrow L_{11}^{-1}$	(trinv1, N, 100, ., 300, 1)
$L_{10} \leftarrow L_{10}L_{00}$	(dtrmm, R, L, N, N, 100, 200, v1, ., 300, ., 300)
$L_{10} \leftarrow -L_{11}^{-1}L_{10}$	(dtrsm, L, L, N, N, 100, 200, v-1, ., 300, ., 300)
$L_{11} \leftarrow L_{11}^{-1}$	(trinv1, N, 100, ., 300, 1)

Table 4.1: Subroutine invocation of dtrsm(N, 300, A, 300, 100).

```

1 [[ 'dtrmm', 'R', 'L', 'N', 'N', 100, 0, 'v1', None, 300, None, 300],
2  [ 'dtrsm', 'L', 'L', 'N', 'N', 100, 0, 'v-1', None, 300, None, 300],
3  [ 'trinv1', 'N', 100, None, 300, 1],
4  [ 'dtrmm', 'R', 'L', 'N', 'N', 100, 100, 'v1', None, 300, None, 300],
5  [ 'dtrsm', 'L', 'L', 'N', 'N', 100, 100, 'v-1', None, 300, None, 300],
6  [ 'trinv1', 'N', 100, None, 300, 1],
7  [ 'dtrmm', 'R', 'L', 'N', 'N', 100, 200, 'v1', None, 300, None, 300],
8  [ 'dtrsm', 'L', 'L', 'N', 'N', 100, 200, 'v-1', None, 300, None, 300],
9  [ 'trinv1', 'N', 100, None, 300, 1]]

```

□

4.2 Triangular Inverse $L \leftarrow L^{-1}$

The first operation we consider is the inversion of a triangular matrix, $L \leftarrow L^{-1}$. In [Section 1.4.1](#), we presented four blocked algorithms that perform this operation with the following update statements:

Variant 1	Variant 2
$L_{10} \leftarrow L_{10}L_{00}$	$L_{21} \leftarrow L_{22}^{-1}L_{21}$
$L_{10} \leftarrow -L_{11}^{-1}L_{10}$	$L_{21} \leftarrow -L_{21}L_{11}^{-1}$
$L_{11} \leftarrow L_{11}^{-1}$	$L_{11} \leftarrow L_{11}^{-1}$
Variant 3	Variant 4
$L_{21} \leftarrow -L_{21}L_{11}^{-1}$	$L_{21} \leftarrow -L_{22}^{-1}L_{21}$
$L_{20} \leftarrow L_{21}L_{10} + L_{20}$	$L_{20} \leftarrow -L_{21}L_{10} + L_{20}$
$L_{10} \leftarrow L_{11}^{-1}L_{10}$	$L_{10} \leftarrow L_{10}L_{00}$
$L_{11} \leftarrow L_{11}^{-1}$	$L_{11} \leftarrow L_{11}^{-1}$

These algorithms' C implementations used for the following performance prediction is given in [Appendix B.1](#).

In [Figure 4.1](#), we present performance measurements of these algorithms with the arguments

```

diag  n  A  ldA  blocksize
trinv1( N , n , A , n , 96 ),

```

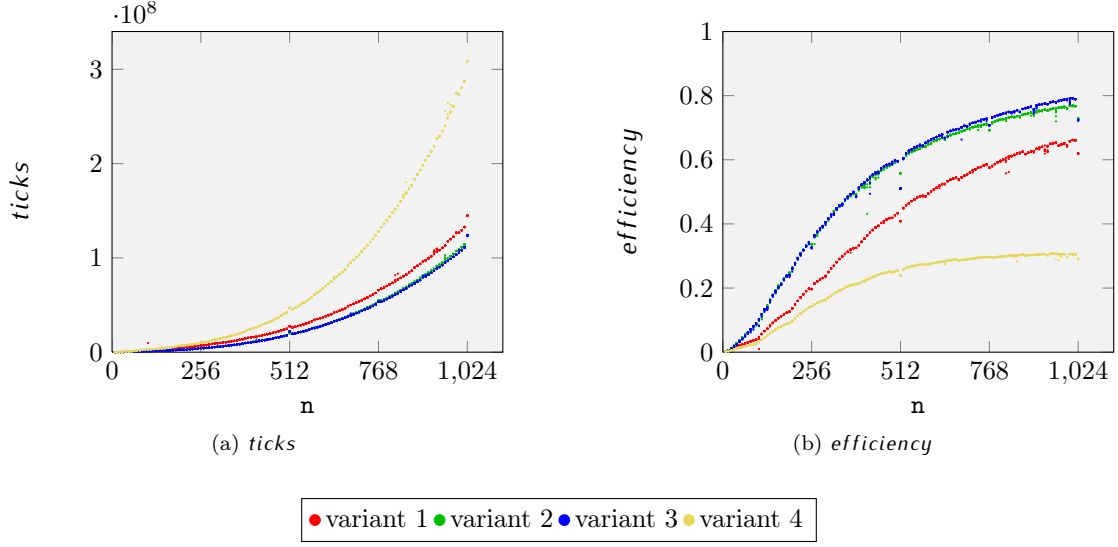


Figure 4.1: *ticks* and *efficiency* for `trinv(N, n, A, n, 96)`.

varying the matrix size $n \in \{8, 16, \dots, 1024\}$, taking 10 measurements at each point. We show the metrics *ticks* and *efficiency*, where

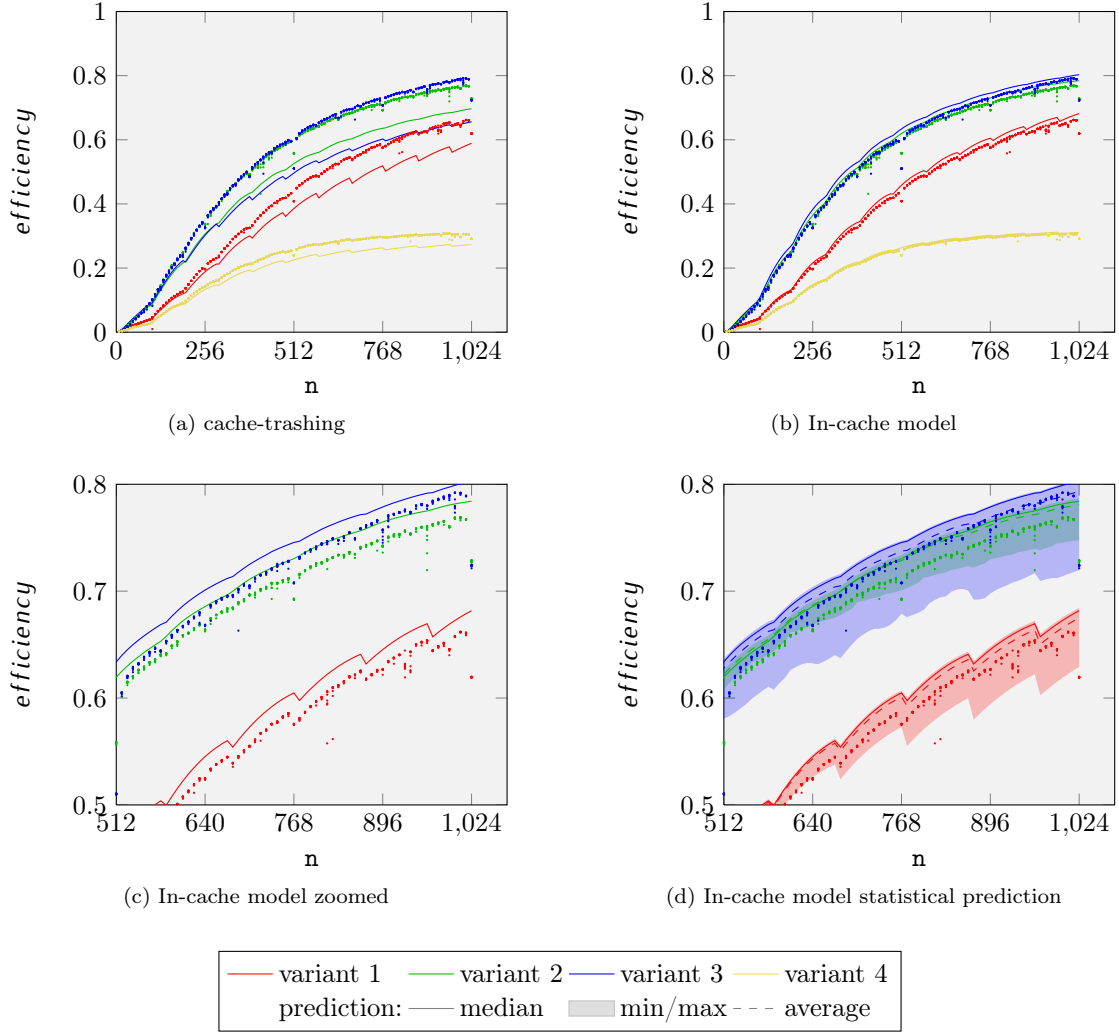
$$efficiency = \frac{\frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n}{2 \cdot ticks}.$$

We now turn to the prediction of these measurements *ticks*, for now, considering the median of our models. For our first performance estimate, we use performance models for `dtrsm`, `dtrmm`, `dgemm`, and the unblocked versions of the blocked algorithms¹. These are generated with a cache-trashing Sampler and the Modeler configuration selected in Section 3.4.2.3. For each algorithm execution, we generate consider the list of subroutine invocations, containing calls to BLAS and the algorithms' unblocked versions. We evaluate the performance models for these routine invocations and accumulate the obtained estimates, resulting in our performance prediction.

Since the *efficiency* graphs for the four algorithms (Figure 4.1b) give a better impression of their performance, we present the *efficiency* computed from our obtained *ticks* estimates in Figure 4.2a. These predictions are already accurate enough to determine that variants 4 (—) and 1 (—) are less efficient than variants 2 (—) and 3 (—). However, in our predictions, variant 2 (—) erroneously becomes more efficient than variant 3 (—) for increasing n ; this is *not* the case in the measurements. Furthermore, all *efficiency* predictions are too low; This results from overestimating *ticks*. This overestimation is due to the memory locality situations used during the generation of our performance models; the matrix arguments were placed in main memory.

For our second set of predictions, we use a performance model generated with an in-cache Sampler configuration. The results are shown in Figure 4.2b; a zoom-in on the upper right part of the plot ($n \geq 512$ and $0.5 \leq efficiency \leq 0.8$) is given in Figure 4.2c. Compared to our previous predictions, these results show a significantly closer to the measurements. Furthermore, all algorithm variants are now ranked correctly.

¹The models for the unblocked versions are limited to values of `size` up to 256 — large enough for the unblocked algorithm invocations.

Figure 4.2: *efficiency* predictions $\text{trinvi}(N, n, A, n, 96)$.

Up to this point, we presented estimates for the median of the performance counter. In Figure 4.2d we additionally take the quantities average, minimum, and maximum into account. The range between minimum and maximum (■) covers almost all measurements and gives a good idea on the expected results; however, they are very broad — they overlap and even include each other. The average (---) is closer to the measured algorithm performance than the previously used median. Relying on the average predictions, however, is dangerous, since they are obtained for models generated with an error bound on the median and are influenced by outliers.

Block-size. So far we predicted performance for varying matrix sizes with highly satisfactory results. We now turn to our second aspect of interest: determining the most efficient block-size.

For this purpose, we now fix the matrix size to² $n = 1016$ and vary the block-size:

² We chose 1016, since we observed outliers for the algorithms' performance measurements at 1024 which we are

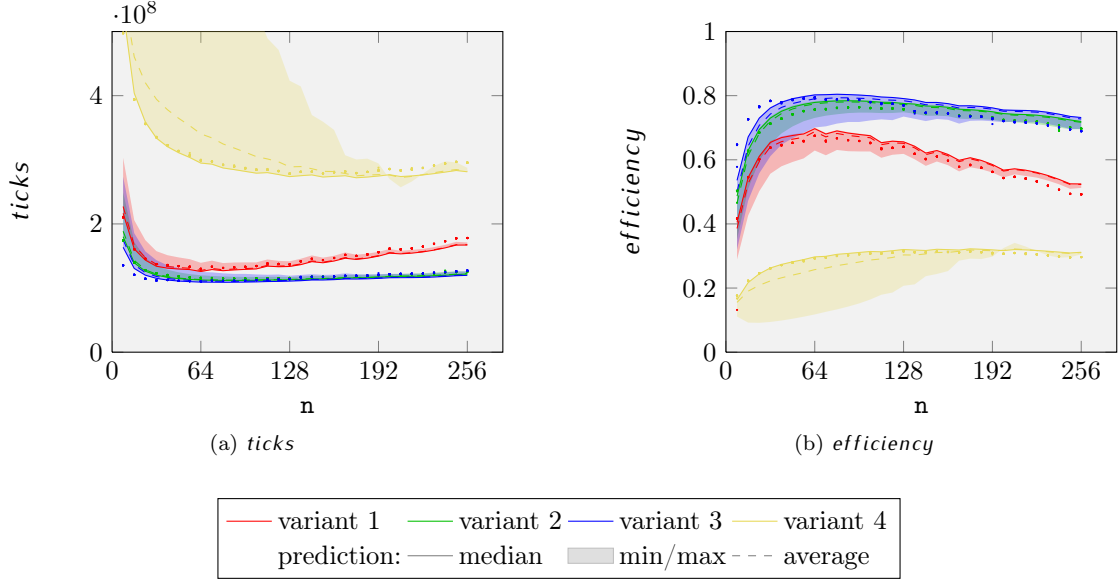


Figure 4.3: *efficiency* predictions `trinv(N, 1016, A, 1016, blocksize)`.

```
diag      n      A      ldA      blocksize
trinv( N , 1016, A , 1016, blocksize).
```

The resulting predictions (Figure 4.3) represent the measurements very well for the most efficient block-sizes between 48 and 128. Variant 3 (—) — the fastest — attains its highest performance for a block-size of 64 both in our prediction and the measurements.

The quality of our prediction decreases for very large and very small block-sizes.

Small block-sizes. The algorithms invoke subroutines very often with long and skinny matrices. For such matrices, BLAS routines usually do not attain high performance; this is carried forward to the blocked algorithms. Additionally, our models are less accurate for small argument values, further decreasing the prediction accuracy.

Large block-size. The unblocked versions of the algorithm increasingly contribute to the overall performance. Since these unblocked versions are less efficient, the overall performance decreases; *ticks* are overestimated only slightly worse than for block-sizes around 100.

For our goal — determining the fastest algorithm configuration — the low accuracy in regions with low performance is not a major problem.

In the predictions of the maximum and average *ticks* for variant 4 (—), we further observe severe inaccuracies for block-sizes between 8 and 160. These are due to outliers in the measurements that the corresponding models were constructed upon. These outliers do not carry forward to the minimum and the median of *ticks*.

4.3 LU Decomposition $LU \leftarrow A$

The second operation we study is the LU decomposition $LU = A$ of a square matrix A . It is very commonly used to solve linear systems $AX = B$: With $LU = A$, we can write this as $LUX = B$

not able to explain.

and apply `dtrsm` (solution of a triangular system) twice to obtain $X = U^{-1}L^{-1}B$.

There are five blocked algorithms for the LU decomposition. They take A as an input and compute L and U in place: The unit-diagonal lower triangular L is stored in the strictly lower triangular part of A , whereas the non-unit-diagonal U is stored in the upper triangular part of A . Within the scheme of the blocked algorithms (Section 1.4), the five algorithmic variants perform the following updates:

Variant 1 $A_{01} \leftarrow (\nabla A_{00})^{-1} A_{01}$ $A_{10} \leftarrow A_{10} (\nabla A_{00})^{-1}$ $A_{11} \leftarrow A_{11} - A_{10} A_{01}$ $A_{11} \leftarrow LU(A_{11})$	Variant 2 $A_{10} \leftarrow A_{10} (\nabla A_{00})^{-1}$ $A_{11} \leftarrow A_{11} - A_{10} A_{01}$ $A_{11} \leftarrow LU(A_{11})$ $A_{12} \leftarrow A_{12} - A_{10} A_{02}$ $A_{12} \leftarrow (\nabla A_{11})^{-1} A_{12}$	Variant 3 $A_{01} \leftarrow (\nabla A_{00})^{-1} A_{01}$ $A_{11} \leftarrow A_{11} - A_{10} A_{01}$ $A_{11} \leftarrow LU(A_{11})$ $A_{21} \leftarrow A_{21} - A_{20} A_{01}$ $A_{21} \leftarrow A_{21} (\nabla A_{11})^{-1}$
Variant 4 $A_{11} \leftarrow A_{11} - A_{10} A_{01}$ $A_{11} \leftarrow LU(A_{11})$ $A_{12} \leftarrow A_{12} - A_{10} A_{02}$ $A_{12} \leftarrow (\nabla A_{11})^{-1} A_{12}$ $A_{21} \leftarrow A_{21} - A_{20} A_{01}$ $A_{21} \leftarrow A_{21} (\nabla A_{11})^{-1}$	Variant 5 $A_{11} \leftarrow LU(A_{11})$ $A_{12} \leftarrow (\nabla A_{11})^{-1} A_{12}$ $A_{21} \leftarrow A_{21} (\nabla A_{11})^{-1}$ $A_{22} \leftarrow A_{22} - A_{21} A_{12}$	

Here, ∇A and ∇A denote the (strictly) lower and upper triangular part of A , respectively; $LU(A)$ denotes the LU decomposition itself — an recursive application of the algorithm with block-size 1 to a submatrix. Our C implementation of these algorithms is given in Appendix B.2; their signatures are `lui(n, A, ldA, blocksize)` with $i \in \{1, 2, 3, 4\}$.

We use the same method applied to $L \leftarrow L^{-1}$ in the previous section to estimate the performance of these algorithms with the arguments

$$\text{lui}(\text{n}, \text{A}, \text{ldA}, \text{blocksize}),$$

where $\mathbf{n} \in \{8, 16, \dots, 1024\}$. The results of these predictions and measurements of the corresponding algorithm executions are shown in Figure 4.4, where *efficiency* is given by

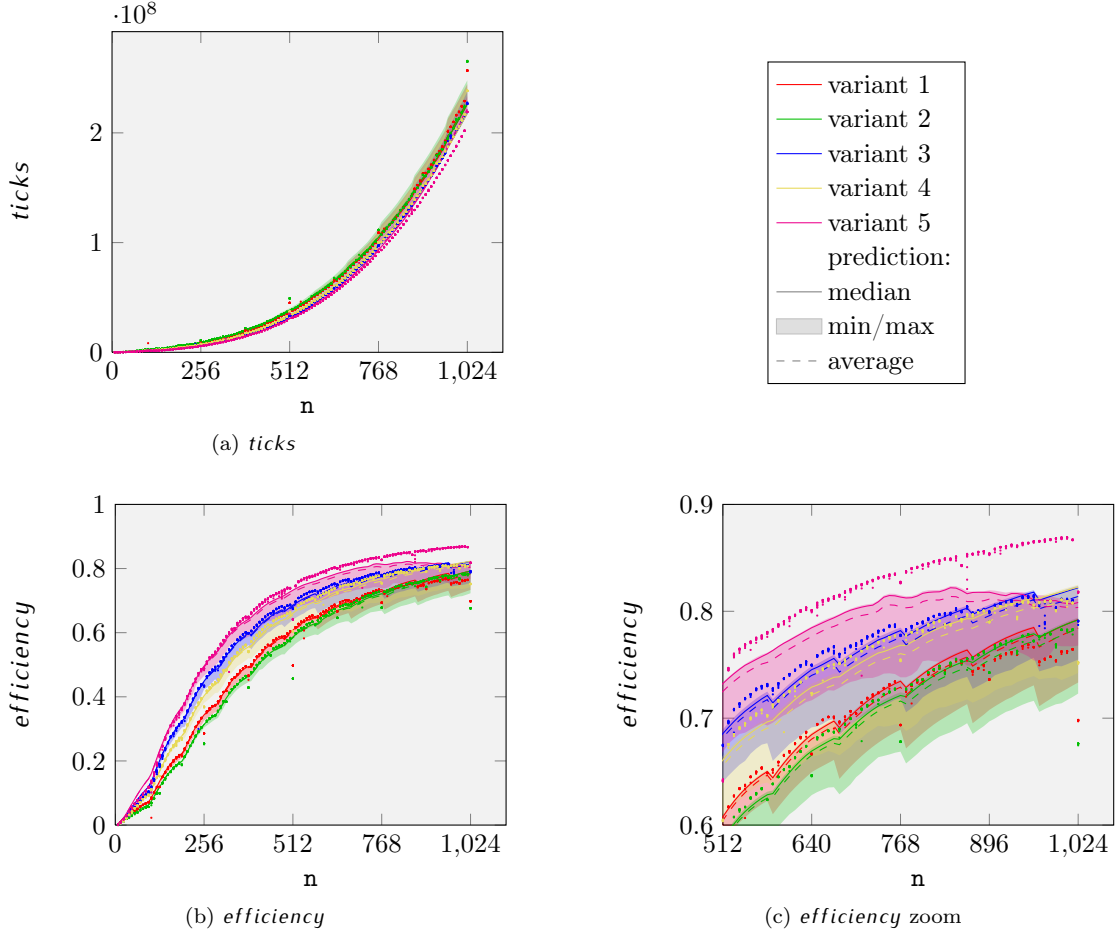
$$\text{efficiency} = \frac{\frac{1}{3}\mathbf{n}^3 + \frac{1}{2}\mathbf{n}^2 - \frac{5}{6}\mathbf{n}}{2\text{ticks}}.$$

Since it is almost impossible to distinguish the performance of the five variants in the *ticks* plot, we focus on *efficiency* (Figures 4.4b and 4.4c). We observe that for the most part our predictions are fairly close to the measurements. Up to $\mathbf{n} \approx 900$, they can be used to rank the five algorithms correctly according to their performance.

However, the performance of variant 5 (—) — the fastest — is predicted with decreasing accuracy for larger values of \mathbf{n} ; the predictions are generally too low and diverge from our measurements. This leads so far, that the algorithms are incorrectly ranked for problems larger than $n = 900$. While the other variants are also slightly underestimated, their predictions are more stable and accurate.

4.4 Sylvester Equation: Solving $LX + XU = C$ for X

We now study of a more complicated operation: the solution of the Sylvester equation. This operation is encountered in control theory and is generally of the form $AX + XB = C$, where

Figure 4.4: *efficiency* predictions for `lvi`(n , A , n , 96).

$A \in \mathbb{R}^{m \times m}$, $B \in \mathbb{R}^{n \times n}$, and $C \in \mathbb{R}^{m \times n}$ are given and $X \in \mathbb{R}^{m \times n}$ is to be computed. We consider a special case of this equation, where A and B are lower and upper triangular, respectively: $LX + XU = C$.

With CLICK [15, 16], a tool for the automatic generation of blocked algorithms, we found 16 algorithmic variants for this operation. Each of them takes three matrices L , U , and X ; X initially contains the input matrix C and is overwritten with the solution X by the algorithms.

The update statements for the 16 algorithms are given below. Within these, $\Omega(L, U, X)$ denotes a recursive invocation of the Sylvester equation solver for a smaller matrix X . The C implementation of the algorithms is given in [Appendix B.3](#); their signature is `sylvi(m, n, L, ldL, U, ldU, X, ldX, blocksize)` with $i \in \{1, \dots, 16\}$.

Variant 1	Variant 2	Variant 3
$X_{01} \leftarrow X_{01} - X_{00}U_{01}$ $X_{10} \leftarrow X_{10} - L_{10}X_{00}$ $X_{01} \leftarrow \Omega(L_{00}, U_{11}, X_{01})$ $X_{10} \leftarrow \Omega(L_{11}, U_{00}, X_{10})$ $X_{11} \leftarrow X_{11} - X_{10}U_{01}$ $X_{11} \leftarrow X_{11} - L_{10}X_{01}$ $X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$	$X_{01} \leftarrow X_{01} - X_{00}U_{01}$ $X_{10} \leftarrow \Omega(L_{11}, U_{00}, X_{10})$ $X_{01} \leftarrow \Omega(L_{00}, U_{11}, X_{01})$ $X_{11} \leftarrow X_{11} - X_{10}U_{01}$ $X_{20} \leftarrow X_{20} - L_{21}X_{10}$ $X_{11} \leftarrow X_{11} - L_{10}X_{01}$ $X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$ $X_{21} \leftarrow X_{21} - L_{21}X_{11}$ $X_{21} \leftarrow X_{21} - L_{20}X_{01}$	$X_{01} \leftarrow X_{01} - X_{00}U_{01}$ $X_{11} \leftarrow X_{11} - X_{10}U_{01}$ $X_{21} \leftarrow X_{21} - X_{20}U_{01}$ $X_{01} \leftarrow \Omega(L_{00}, U_{11}, X_{01})$ $X_{11} \leftarrow X_{11} - L_{10}X_{01}$ $X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$ $X_{21} \leftarrow X_{21} - L_{21}X_{11}$ $X_{21} \leftarrow X_{21} - L_{20}X_{01}$ $X_{21} \leftarrow \Omega(L_{22}, U_{11}, X_{21})$

Variant 4	Variant 5	Variant 6
$X_{01} \leftarrow X_{01} - X_{00}U_{01}$ $X_{12} \leftarrow X_{12} - X_{10}U_{02}$ $X_{01} \leftarrow \Omega(L_{00}, U_{11}, X_{01})$ $X_{11} \leftarrow X_{11} - L_{10}X_{01}$ $X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$ $X_{21} \leftarrow X_{21} - L_{21}X_{11}$ $X_{21} \leftarrow X_{21} - L_{20}X_{01}$ $X_{21} \leftarrow \Omega(L_{22}, U_{11}, X_{21})$ $X_{22} \leftarrow X_{22} - X_{21}U_{12}$	$X_{01} \leftarrow \Omega(L_{00}, U_{11}, X_{01})$ $X_{10} \leftarrow X_{10} - L_{10}X_{00}$ $X_{02} \leftarrow X_{02} - X_{01}U_{12}$ $X_{10} \leftarrow \Omega(L_{11}, U_{00}, X_{10})$ $X_{11} \leftarrow X_{11} - X_{10}U_{01}$ $X_{11} \leftarrow X_{11} - L_{10}X_{01}$ $X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$ $X_{12} \leftarrow X_{12} - X_{11}U_{12}$ $X_{12} \leftarrow X_{12} - X_{10}U_{02}$	$X_{01} \leftarrow \Omega(L_{00}, U_{11}, X_{01})$ $X_{10} \leftarrow \Omega(L_{11}, U_{00}, X_{10})$ $X_{02} \leftarrow X_{02} - X_{01}U_{12}$ $X_{11} \leftarrow X_{11} - X_{10}U_{01}$ $X_{20} \leftarrow X_{20} - L_{21}X_{10}$ $X_{11} \leftarrow X_{11} - L_{10}X_{01}$ $X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$ $X_{12} \leftarrow X_{12} - X_{11}U_{12}$ $X_{21} \leftarrow X_{21} - L_{21}X_{11}$ $X_{12} \leftarrow X_{12} - X_{10}U_{02}$ $X_{21} \leftarrow X_{21} - L_{20}X_{01}$

Variant 7	Variant 8	Variant 9
$X_{01} \leftarrow \Omega(L_{00}, U_{11}, X_{01})$ $X_{11} \leftarrow X_{11} - X_{10}U_{01}$ $X_{21} \leftarrow X_{21} - X_{20}U_{01}$ $X_{02} \leftarrow X_{02} - X_{01}U_{12}$ $X_{11} \leftarrow X_{11} - L_{10}X_{01}$ $X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$ $X_{12} \leftarrow X_{12} - X_{11}U_{12}$ $X_{21} \leftarrow X_{21} - L_{21}X_{11}$ $X_{12} \leftarrow X_{12} - X_{10}U_{02}$ $X_{21} \leftarrow X_{21} - L_{20}X_{01}$ $X_{21} \leftarrow \Omega(L_{22}, U_{11}, X_{21})$	$X_{01} \leftarrow \Omega(L_{00}, U_{11}, X_{01})$ $X_{02} \leftarrow X_{02} - X_{01}U_{12}$ $X_{11} \leftarrow X_{11} - L_{10}X_{01}$ $X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$ $X_{12} \leftarrow X_{12} - X_{11}U_{12}$ $X_{21} \leftarrow X_{21} - L_{21}X_{11}$ $X_{21} \leftarrow X_{21} - L_{20}X_{01}$ $X_{21} \leftarrow \Omega(L_{22}, U_{11}, X_{21})$ $X_{22} \leftarrow X_{22} - X_{21}U_{12}$	$X_{10} \leftarrow X_{10} - L_{10}X_{00}$ $X_{10} \leftarrow \Omega(L_{11}, U_{00}, X_{10})$ $X_{11} \leftarrow X_{11} - X_{10}U_{01}$ $X_{11} \leftarrow X_{11} - L_{10}X_{01}$ $X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$ $X_{12} \leftarrow X_{12} - X_{11}U_{12}$ $X_{12} \leftarrow X_{12} - X_{10}U_{02}$ $X_{12} \leftarrow X_{12} - L_{10}X_{02}$ $X_{12} \leftarrow \Omega(L_{11}, U_{22}, X_{12})$

Variant 10
$X_{10} \leftarrow X_{10} - L_{10}X_{00}$
$X_{21} \leftarrow X_{21} - L_{20}X_{01}$
$X_{10} \leftarrow \Omega(L_{11}, U_{00}, X_{10})$
$X_{11} \leftarrow X_{11} - X_{10}U_{01}$
$X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$
$X_{12} \leftarrow X_{12} - X_{11}U_{12}$
$X_{12} \leftarrow X_{12} - X_{10}U_{02}$
$X_{12} \leftarrow \Omega(L_{11}, U_{22}, X_{12})$
$X_{22} \leftarrow X_{22} - L_{21}X_{12}$

Variant 11
$X_{10} \leftarrow \Omega(L_{11}, U_{00}, X_{10})$
$X_{11} \leftarrow X_{11} - X_{10}U_{01}$
$X_{20} \leftarrow X_{20} - L_{21}X_{10}$
$X_{11} \leftarrow X_{11} - L_{10}X_{01}$
$X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$
$X_{12} \leftarrow X_{12} - X_{11}U_{12}$
$X_{21} \leftarrow X_{21} - L_{21}X_{11}$
$X_{12} \leftarrow X_{12} - X_{10}U_{02}$
$X_{21} \leftarrow X_{21} - L_{20}X_{01}$
$X_{12} \leftarrow X_{12} - L_{10}X_{02}$
$X_{12} \leftarrow \Omega(L_{11}, U_{22}, X_{12})$

Variant 12
$X_{10} \leftarrow \Omega(L_{11}, U_{00}, X_{10})$
$X_{11} \leftarrow X_{11} - X_{10}U_{01}$
$X_{20} \leftarrow X_{20} - L_{21}X_{10}$
$X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$
$X_{12} \leftarrow X_{12} - X_{11}U_{12}$
$X_{21} \leftarrow X_{21} - L_{21}X_{11}$
$X_{12} \leftarrow X_{12} - X_{10}U_{02}$
$X_{12} \leftarrow \Omega(L_{11}, U_{22}, X_{12})$
$X_{22} \leftarrow X_{22} - L_{21}X_{12}$

Variant 13
$X_{11} \leftarrow X_{11} - X_{10}U_{01}$
$X_{21} \leftarrow X_{21} - X_{20}U_{01}$
$X_{11} \leftarrow X_{11} - L_{10}X_{01}$
$X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$
$X_{12} \leftarrow X_{12} - X_{11}U_{12}$
$X_{21} \leftarrow X_{21} - L_{21}X_{11}$
$X_{12} \leftarrow X_{12} - X_{10}U_{02}$
$X_{21} \leftarrow X_{21} - L_{20}X_{01}$
$X_{12} \leftarrow X_{12} - L_{10}X_{02}$
$X_{21} \leftarrow \Omega(L_{22}, U_{11}, X_{21})$
$X_{12} \leftarrow \Omega(L_{11}, U_{22}, X_{12})$

Variant 14
$X_{11} \leftarrow X_{11} - X_{10}U_{01}$
$X_{21} \leftarrow X_{21} - X_{20}U_{01}$
$X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$
$X_{12} \leftarrow X_{12} - X_{11}U_{12}$
$X_{21} \leftarrow X_{21} - L_{21}X_{11}$
$X_{12} \leftarrow X_{12} - X_{10}U_{02}$
$X_{21} \leftarrow \Omega(L_{22}, U_{11}, X_{21})$
$X_{12} \leftarrow \Omega(L_{11}, U_{22}, X_{12})$
$X_{22} \leftarrow X_{22} - L_{21}X_{12}$

Variant 15
$X_{11} \leftarrow X_{11} - L_{10}X_{01}$
$X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$
$X_{12} \leftarrow X_{12} - X_{11}U_{12}$
$X_{21} \leftarrow X_{21} - L_{21}X_{11}$
$X_{12} \leftarrow X_{12} - L_{10}X_{02}$
$X_{21} \leftarrow X_{21} - L_{20}X_{01}$
$X_{12} \leftarrow \Omega(L_{11}, U_{22}, X_{12})$
$X_{21} \leftarrow \Omega(L_{22}, U_{11}, X_{21})$
$X_{22} \leftarrow X_{22} - X_{21}U_{12}$

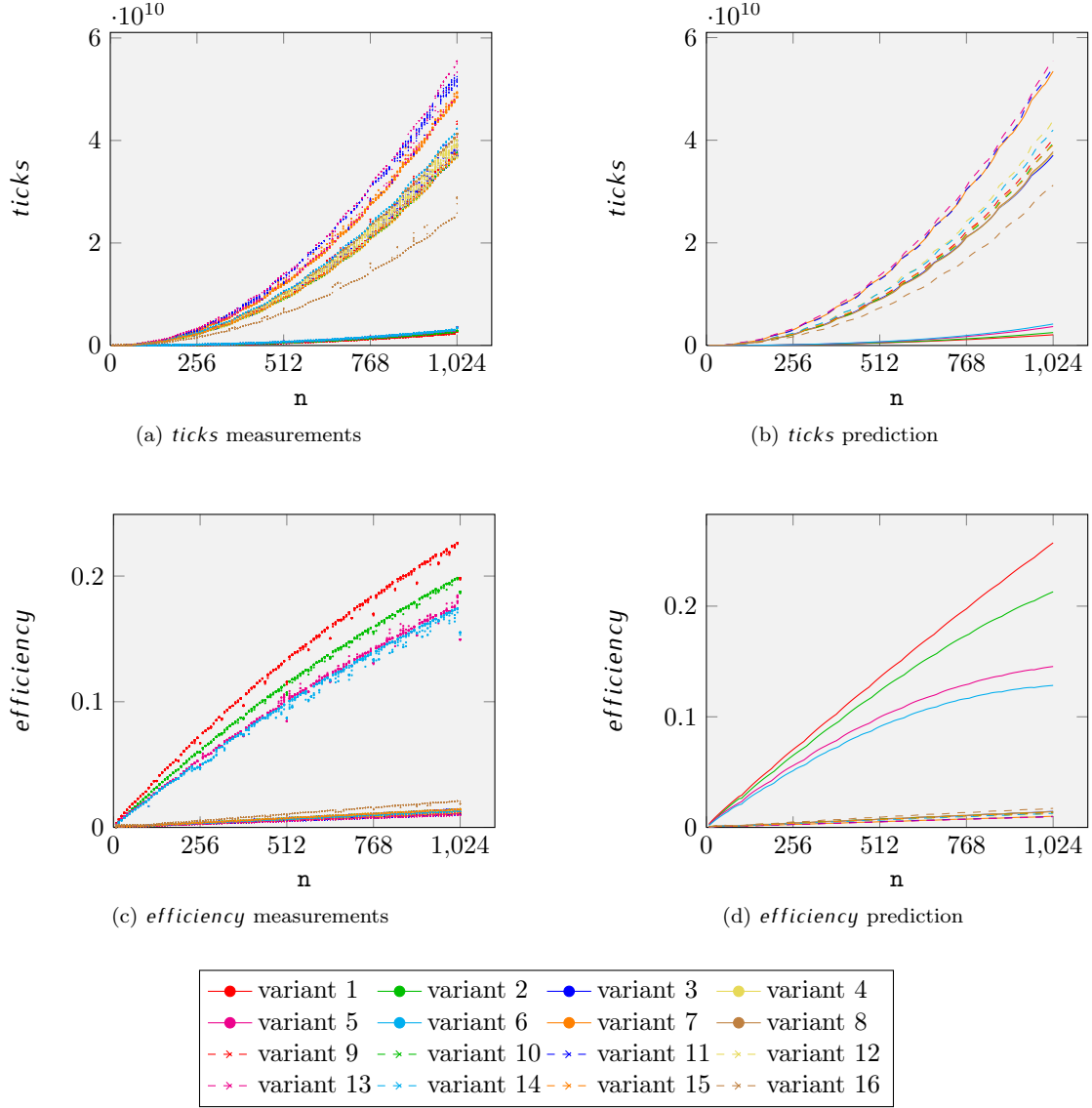
Variant 16
$X_{11} \leftarrow \Omega(L_{11}, U_{11}, X_{11})$
$X_{12} \leftarrow X_{12} - X_{11}U_{12}$
$X_{21} \leftarrow X_{21} - L_{21}X_{11}$
$X_{12} \leftarrow \Omega(L_{11}, U_{22}, X_{12})$
$X_{21} \leftarrow \Omega(L_{22}, U_{11}, X_{21})$
$X_{22} \leftarrow X_{22} - X_{21}U_{12}$
$X_{22} \leftarrow X_{22} - L_{21}X_{12}$

These algorithms differ in a few ways from those in the previous sections:

- They operate on three matrices, overwriting one of them with the output.
- The input matrices are of different sizes: $L \in \mathbb{R}^{m \times n}$, $U \in \mathbb{R}^{n \times n}$, and $X \in \mathbb{R}^{m \times n}$. The matrices are traversed along the diagonal as far as possible and then along the remaining dimension (see [Section 1.4.2](#)).
- There are three recursive calls Ω in each step of the matrix traversal. These operate not only on the $X_{11} \in \mathbb{R}^{\text{blocksize} \times \text{blocksize}}$ but also on the matrix panels X_{01} , X_{10} , X_{12} , and X_{21} . For the latter, our C implementation invokes the blocked algorithms recursively; only the small matrices X_{11} trigger their unblocked versions.

For our performance predictions, we focus on the case

$$\text{sylv}i(\overset{m}{n}, \overset{n}{n}, \overset{L}{L}, \overset{\text{ldL}}{n}, \overset{U}{U}, \overset{\text{ldU}}{n}, \overset{X}{X}, \overset{\text{ldX}}{n}, \overset{\text{blocksize}}{96}).$$

Figure 4.5: *efficiency* predictions for $\text{sylvi}(n, n, L, n, U, n, X, n, 96)$.

All matrices are of size $n \times n$, $n \in \{8, 16, \dots, 1024\}$ and we use the block-size 96. Figure 4.5 compares our predictions for these algorithms with corresponding measurements of their implementations, where

$$\text{efficiency} = \frac{n^3 + n^2}{2\text{ticks}}.$$

We observe significantly different performances across algorithms: At $n = 1024$ variant 1 (—●—) is 20 times faster than variant 13 (---×---). In the *ticks* plots, we see that our predictions separate the fast and slow variants very well. To rank the top candidates, we turn to *efficiency*; here, we see that the prediction quality is not as good as in our previous studies. However, the algorithms are

correctly ranked.

Chapter 5

Conclusion and Future Work

In this thesis, we have introduced methods and tools to analyze and model the performance of dense linear algebra routines. Our goal was to rank sets of blocked algorithms according to their performance without executing them. Towards this goal, we created a set of automatic performance analysis and modeling tools. The Sampler measures the performance of routine executions; based on this tool, the Modeler generates performance models for specified routines. With these models, we were able to accurately predict the performance of blocked algorithms, rank them correctly, and determine the optimal algorithmic block-size.

Sampler. As a base for our performance analysis, we designed a performance measurement tool: the Sampler (Section 2.3). This flexible tool can sample any dense linear algebra routine execution on any system. Provided with library or object files for a set of routines and according header files, its build system automatically generates an executable that allows to measure the performance of these routines.

The cycle-accurate time stamp counter (RDTSC) serves as the base of its execution time measurements *ticks*. The Performance Application Programming Interface (PAPI) is optionally¹ used to access a wide range of other performance counters, such as *flops* or *L1misses*. These performance counters and further properties of the Sampler, such as memory handling policies, are specified through a simple configuration file (Section 2.3.2.1).

The Sampler can be interfaced with other programs through its standard input and output streams. On its input stream, it expects a series of sampling requests, consisting of routine names and argument values; on the output stream, it provides performance measurements for these routine invocations.

Thanks to its flexibility, functionalities, and simple interface, the Sampler serves as an ideal base for performance analysis in dense linear algebra. Apart from performance modeling, it can be used for tasks such as performance debugging or tuning.

Modeler. Based on the Sampler, we constructed an automatic performance modeling tool: the Modeler (Section 3.3). For dense linear algebra routines, this tool builds models that represent the routines' performance as a function of a set of discrete (e.g., *side*, *transA*) and continuous (e.g., *m*, *n*) arguments. The generated models yield probabilistic performance estimates in the form of statistical quantities, such as minimum, median, or average.

The Modeler, written in Python, is highly flexible: It provides a configuration system to specify a wide range of settings. These include but are not limited to the following:

¹ requiring a kernel patch.

- The choice of the Sampler instance and its configuration;
- The routines to be modeled;
- The treatment of the routine arguments;
- The performance counters to be modeled;
- The type of polynomial modeling to be used;
- The accuracy and resolution of the polynomial models.

The Modeler aims at generating highly accurate performance models, while at the same time using as few measurements as possible. This poses a trade-off: reducing the number of samples generally decreases the model accuracy and vice versa. The Modeler configuration allows to balance these two aspects according to the desires of the user (see [Section 3.4.2](#)).

Ranking. With the models automatically generated by the Modeler, we can predict the performance of blocked algorithms execution-less ([Chapter 4](#)). Our accurate predictions allow us to

- Rank several algorithms according to their performance, and
- Determine the optimal algorithmic block-size.

5.1 Outlook

The work presented in this thesis offers a number of possible directions for further research.

Prediction of Other Performance Metrics. In our study of blocked algorithms, we focused on predicting the execution time *ticks* — one of many performance metrics. Our models, however, can describe a large variety of performance counters, all of which can be included in the prediction process. Of particular interest might be the cache miss counters *L1misses* through *L3misses*, since these are closely related to the processor’s energy consumption.

Other Algorithm Types. We limited ourselves to the performance prediction of blocked algorithms. These, however, are only one type of algorithms used in dense linear algebra. Other algorithm types include algorithms-by-block and recursive algorithms. Like blocked algorithms, these are based on BLAS Level-3 routines; our models for these routines can be used in the performance prediction of these algorithms.

Shared Memory Parallelism. We were concerned with analyzing the performance of single core algorithms. It is crucial to understand the phenomena and behaviors that appear in this scenario before we target parallel algorithms. Since in the field of high performance computing CPUs usually provide several cores, considering shared memory parallelism is a natural next step.

For blocked algorithms, parallelism can be exploited through multi-threaded BLAS implementations. A possible approach would be to apply our measurement and modeling framework to these parallel BLAS routines and rank the algorithms accordingly.

Extrapolation. In this thesis, we build performance models for routines that represent a certain range of size arguments. We then use these models to predict the performance of blocked algorithms within the same range. In principle, the polynomials of our models can be extrapolated to estimate the performance of routine executions beyond their scope. Designed for a limited range of argument values, the estimation quality of our models will deteriorate when we extrapolate. However, extrapolation is an interesting usage scenario of performance models and performance modeling in general.

Bibliography

- [1] ATLAS homepage. <http://math-atlas.sourceforge.net/>.
- [2] BLAS reference implementation. <http://www.netlib.org/blas/>.
- [3] Gotoblas. <http://www.tacc.utexas.edu/tacc-projects/gotoblas2>.
- [4] Intel® Compilers. <http://software.intel.com/en-us/articles/intel-compilers/>.
- [5] Intel® Math Kernel Library. <http://software.intel.com/en-us/articles/intel-mkl/>.
- [6] Intel® Xeon Processor E5450. [http://ark.intel.com/products/33083/Intel-Xeon-Processor-E5450-\(12M-Cache-3_00-GHz-1333-MHz-FSB\)](http://ark.intel.com/products/33083/Intel-Xeon-Processor-E5450-(12M-Cache-3_00-GHz-1333-MHz-FSB)).
- [7] LAPACK – Linear Algebra PACKage. www.netlib.org/lapack/.
- [8] Quad-Core AMD Opteron™ 8356. <http://products.amd.com/pages/OpteronCPUDetail.aspx?id=420&f1=&f2=&f3=Yes&f4=&f5=&f6=&f7=&f8=&f9=&f10=&f11=&>.
- [9] ScaLAPACK – Scalable Linear Algebra PACKage. <http://www.netlib.org/scalapack/>.
- [10] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
- [11] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.
- [12] Javier Cuenca, Domingo Giménez, and José González. Architecture of an automatically tuned linear algebra library. *Parallel Comput.*, 30(2):187–210, February 2004.
- [13] J. Dongarra, Jerney Du Cruz, Sven Hammerling, and I. S. Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw.*, 16(1):18–28, March 1990.
- [14] Jack Dongarra and Piotr Luszczek. Reducing the time to tune parallel dense linear algebra routines with partial execution and performance modelling. Technical report, University of Tennessee Computer Science Technical Report, 2010.
- [15] Diego Fabregat-Traver and Paolo Bientinesi. Automatic generation of loop-invariants for matrix operations. In *Computational Science and its Applications, International Conference*, pages 82–92, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

- [16] Diego Fabregat-Traver and Paolo Bientinesi. Knowledge-based automatic generation of partitioned matrix expressions. In Vladimir Gerdt, Wolfram Koepf, Ernst Mayr, and Evgenii Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, volume 6885 of *Lecture Notes in Computer Science*, pages 144–157. Springer Berlin / Heidelberg, 2011.
- [17] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.
- [18] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35(1):4:1–4:14, July 2008.
- [19] Roman Iakymchuk and Paolo Bientinesi. Modeling performance through memory-stalls. *ACM SIGMETRICS Performance Evaluation Review*, 40(2), 2012. To appear.
- [20] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.
- [21] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.
- [22] Antoine Petitet, Clint Whaley, Jack Dongarra, and Andy Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. <http://www.netlib.org/benchmark/hpl/>.
- [23] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Super-Computing 1998: High Performance Networking and Computing*, 1998.
- [24] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. <http://www.cs.utsa.edu/~whaley/papers/spercw04.ps>.

List of Figures

1.1	Inversion of a triangular matrix: execution time and efficiency.	2
1.2	Triangular Inverse — traversal of L	4
1.3	Blocked algorithms — update and repartitioning for a square matrix.	6
1.4	Blocked algorithms — update and repartitioning for a non-square matrix.	7
2.1	Repeated execution of <code>dtrsm</code>	13
2.2	Modifications of the experiment setup for <code>dtrsm</code>	14
3.1	Dependence of <i>ticks</i> on discrete arguments in <code>dtrsm</code>	21
3.2	Dependence of <i>ticks</i> and <i>efficiency</i> on size arguments in <code>dgemm</code>	22
3.3	Dependence of <i>ticks</i> and <i>efficiency</i> on size arguments in <code>dgemm</code> at small scale. . .	23
3.4	Dependence of <i>ticks</i> on scalar arguments in <code>dgemm</code>	24
3.5	Dependence of <i>ticks</i> and <i>L1misses</i> on leading dimension arguments in <code>dgemm</code>	25
3.6	Structure and categorization of performance models and their evaluation.	28
3.7	Translation of the least squares problem.	34
3.8	Choice of sampling points for Model Expansion.	36
3.9	Basic concept of region generation in Model Expansion.	38
3.10	Generation of new regions' base points S	39
3.11	Sampling points distributed on a rectangular grid.	41
3.12	Adaptive Refinement example.	41
3.13	<i>flops</i> model for <code>dtrsm</code> with Model Expansion.	43
3.14	<i>flops</i> model for <code>dtrsm</code> with Adaptive Refinement — Regions for <code>side = L</code>	44
3.15	<i>ticks</i> model for <code>dtrsm</code> with Model Expansion.	45
3.16	<i>ticks</i> model for <code>dtrsm</code> with Adaptive Refinement.	46
3.17	Comparison of modeling methods for <i>ticks</i>	48
4.1	<i>ticks</i> and <i>efficiency</i> for <code>trinv</code> ($N, n, A, n, 96$).	52
4.2	<i>efficiency</i> predictions <code>trinv</code> ($N, n, A, n, 96$).	53
4.3	<i>efficiency</i> predictions <code>trinv</code> ($N, 1016, A, 1016, \text{blocksize}$).	54
4.4	<i>efficiency</i> predictions for <code>lu</code> ($n, A, n, 96$).	56
4.5	<i>efficiency</i> predictions for <code>sylv</code> ($n, n, L, n, U, n, X, n, 96$).	59
A.1	$A \in \mathbb{R}^{m \times n}$ as part of an $M \times N$ matrix.	72
A.2	Performance of <code>dgemm</code> in different BLAS implementations.	73

Appendix A

Introduction to BLAS

In this appendix, we discuss the Basic Linear Algebra Subprograms (BLAS) and their usage. BLAS as a low level interface specification, which can be accessed directly from within C or Fortran. However, virtually all dense linear algebra libraries in any other programming language is built on top of BLAS.

BLAS is separated into three levels:

- BLAS Level-1 routines implement vector operations (e.g., scaling or addition of vectors);
- BLAS Level-2 routines implement matrix-vector operations (e.g., matrix-vector multiplication or solution of a triangular linear system);
- BLAS Level-3 routines implement matrix-matrix operations (e.g., matrix-matrix multiplication or solution of a triangular linear system with multiple right hand sides).

Both BLAS Level-1 and BLAS Level-2 operations are memory bound; they cannot possibly reach the theoretical peak performance of a CPU. Only BLAS Level-3 operations are compute bound; carefully implemented, they can reach up to 99% of the peak performance.

We focus on discussing the following properties of the interfaces for C and Fortran:

- The routine names ([Section A.1](#));
- The routine arguments and matrix storage ([Section A.2](#));
- Routine invocation and results ([Section A.3](#));
- Available implementation ([Section A.4](#)).

Throughout this appendix, we consider `dtrsm` as an exemplary BLAS routine. It computes the solution of a triangular linear system with multiple right hand sides, $B \leftarrow \alpha A^{-1}B$.

A.1 Routine names

Due to restrictions in early versions of Fortran, BLAS routine names consist of a maximum of 6 characters. They are chosen such that they represent both the operation and the data types.

We now study the routine name `dtrsm` and show why it corresponds to the operation $B \leftarrow \alpha A^{-1}B$. Character by character, `dtrsm` has the following meaning:

- Together, `dtrsm` stands for:

Most BLAS Level-2 and Level-3 routines follow this naming scheme; BLAS Level-1 operations use other similarly representative routine names. Examples of other routine names and the corresponding operations are the following:

- Since Fortran appends an underscore `_` to all function names in its object files, this underscore needs to be appended to all BLAS routines, when they are used in C (e.g., `dtrsm` is available as `dtrsm_`).

A.2 Arguments and Matrix Storage

All arguments of BLAS routines are passed by reference, that is, in the form of pointers. This again is necessary for compatibility with Fortran, where arguments are always passed by reference.

For each BLAS routine, we distinguish three groups of arguments: discrete, size, and data arguments. For example, `dtrsm` has the following signature:

$$\text{dtrsm}(\underbrace{\text{side, uplo, transA, diag}}_{\text{discrete}}, \underbrace{\text{m, n}}_{\text{size}}, \underbrace{\alpha}_{\text{scalar}}, \underbrace{\text{A, ldA}}_{\text{matrix}}, \underbrace{\text{B, ldB}}_{\text{matrix}}).$$

α A B
 \nwarrow \nwarrow \nwarrow
 scalar matrix leading
 dimension

- The *discrete arguments* `side`, `uplo`, `transA`, and `diag` specify the exact operation:
 - `side` $\in \{\text{Left}, \text{Right}\}$ identify on which side of B A appears: $B \leftarrow \alpha A^{-1}B$ or $B \leftarrow \alpha B A^{-1}$.
 - `uplo` $\in \{\text{Lowertriangular}, \text{Uppertriangular}\}$ specifies whether A is lower or upper triangular. In operations involving symmetric matrices, `uplo` indicates, which part of the matrix is stored in memory — the lower or upper triangular part.
 - `transA` $\in \{\text{Notranspose}, \text{Transpose}\}$ indicates if the system is solved with A or A^T . For complex valued routines, `Transpose` is replaced by `Complex-conjugate`.
 - `diag` $\in \{\text{Non-Unittriangular}, \text{Unittriangular}\}$ declares whether the entries of the diagonal of A are to be treated as ones (`Unit triangular`) independent of the data stored in memory. Using this argument, it is possible to store both an upper triangular and a lower triangular matrix in the memory of a full matrix, when one of them is unit triangular.

The values of discrete arguments are identified by their first character (e.g., `side = L` for `Left`). All 16 value combinations for the four discrete argument in `dtrsm` are allowed.

- The *size arguments* `m` and `n` specify the sizes of the matrix (and vector) operands. For `dtrsm`, we have $B \in \mathbb{R}^{m \times n}$; depending on the value of `side`, A is of size $m \times m$ (L) or $n \times n$ (R).
- The *data arguments* `alpha`, `A`, `ldA`, `B`, and `ldB` determine the storage locations of the operands:
 - The *scalar argument* `alpha` is (a pointer to) the value of α .
 - The *matrix arguments* `A` and `B` are (pointers to) these matrices in memory. As in Fortran, matrices are expected to be stored in column-major format: the columns of the matrix are continuous in memory; matrices consist of a set of consecutive columns.
 - The *leading dimension* arguments `ldA` and `ldB`, immediately following the corresponding matrix arguments `A` and `B`, give the distance between two adjacent matrix entries in the same row. [Figure A.1](#) shows how this can be used to address a matrix as part of a larger matrix.

For vector-operations there are two further argument types:

- *Vector arguments* (e.g., `x`) are (pointers to) vector in memory.

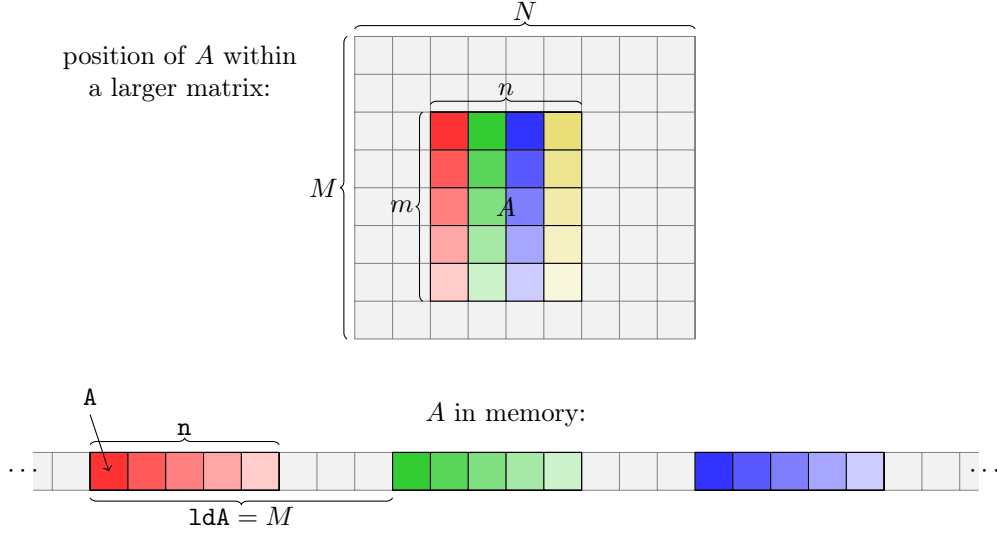


Figure A.1: $A \in \mathbb{R}^{m \times n}$ as part of an $M \times N$ matrix.

- *Increment arguments* (e.g., `incx`) immediately follow the vector arguments and specify the distance between two consecutive elements of a vector in memory. These arguments allow to access both columns and rows of matrices as vectors.

The order of data arguments corresponds to the order of the corresponding operands in the basic form of the operation; for `dtrsm`, `alpha`, `A`, `ldA`, `B`, and `ldB` are in the order of $B \leftarrow \alpha A^{-1} B$.

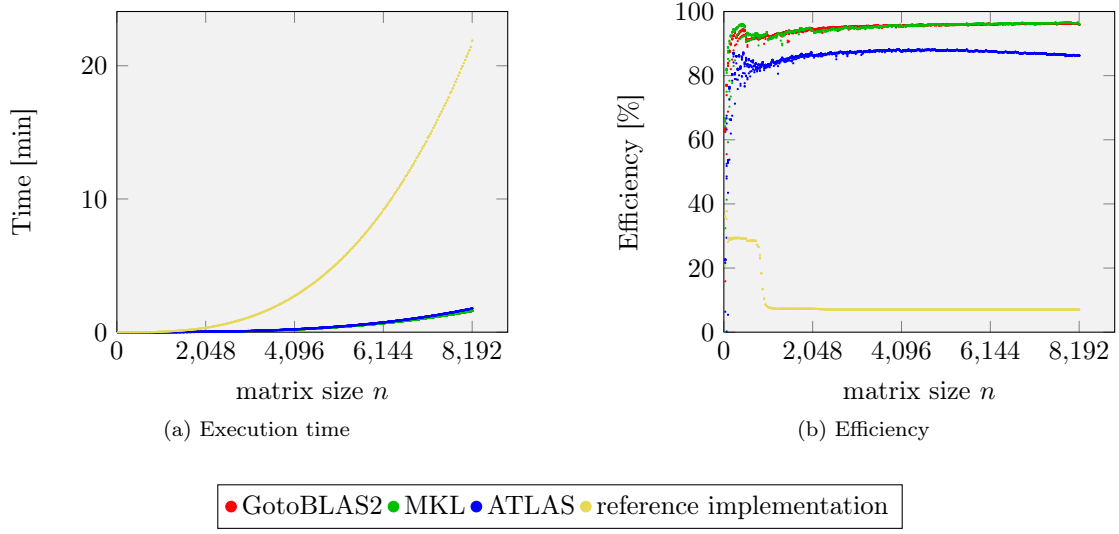
A.3 Routine Invocation and Results

Most BLAS routines do not have an explicit return values, like functions do. Instead, one of the routine arguments is overwritten with the result of the operation — usually the last one. Exceptions are BLAS routines that compute scalar quantities; for example, the inner product of two vectors `ddot` returns $x^T y$.

A.4 Implementations

Since BLAS was introduced in 1979 [20], numerous implementations were developed. In this section, we will list the most prominent implementations for x86 processors and compare their efficiencies.

- The *reference implementation* [2] is a minimal implementation of BLAS. Its purpose is to serve as a reference for both BLAS developers and users. The routines are well documented and easy to read and understand. Since this implementation was not designed with performance in mind, it is not suitable for scientific codes due to its poor performance.
- *GotoBLAS2* [18, 3] is an open source implementation developed at the Texas Advanced Computing Center (University of Texas at Austin); its name stems from its initiator and

Figure A.2: Performance of `dgemm` in different BLAS implementations.

former developer Kazushige Goto. It is written in C and assembly and contains highly optimized kernels for many CPU architectures.

- The *Automatically Tuned Linear Algebra Software* (ATLAS) [23, 24, 1] is another open source implementation. During its building process, this library optimizes itself for the CPU architecture; for this, it uses empirical techniques to estimate the best configuration of its routines.
- Intel’s commercial *Math Kernel Library* (MKL) [5] offers (among other functionality) a high performance implementation of BLAS for Intel architectures.

We now compare the performance of these BLAS implementations by considering `dgemm` ($C \leftarrow \alpha AB + \beta C$). This routine is usually *the* most optimized routine in any high performance library (In many implementations, other BLAS Level-3 routines are based on `dgemm` kernels [17]). We use the Sampler (Section 2.3) on an Intel Harpertown E5450 processor [6] running at 2.99GHz to measure the *ticks* (execution time) of `dgemm` with the following parameters:

```
transA transB m n k alpha A ldA B ldB beta C ldC
dgemm( N      N      m, n, k, 0.5, A, m, B, k, 0.5, C, m ),
```

that is, $C \leftarrow 0.5AB + 0.5C$ with $A, B, C \in \mathbb{R}^{n \times n}$. In Figure A.2 we show the *ticks* and *efficiency* of these executions, where

$$efficiency = \frac{n^3 + 2n^2}{2ticks}.$$

As expected, the reference implementation (●) attains the poorest performance. For matrices that fit in the 12MB cache of the processor, it reaches 28% of the peak performance; for larger matrices, the performance drops rapidly to 7%. The high performance implementations GotoBLAS2 (●), MKL (●), and ATLAS (●) attain efficiencies that exceed the reference implementation by more than a factor of 10. With an efficiencies around 96%, the hand optimized GotoBLAS2 (●) and MKL (●) outperform the automatically tuned ATLAS (●), which reaches 87%.

Appendix B

Implementations of Blocked Algorithms

This appendix lists the implementations of the blocked algorithms used throughout the thesis. All algorithms are written in C and need to be linked with a BLAS library.

We implemented only the blocked version of each algorithm. The unblocked version is implicitly given by passing `blocksize = 1`.

B.1 Triangular Inverse $L \leftarrow L^{-1}$

Listing B.1: `trinv.c` — inversion of a triangular matrix.

```
1 extern void dtrmm_(char*, char*, char*, char*, int*, int*, double*, double*, int*,
   double*, int*);
2 extern void dtrsm_(char*, char*, char*, char*, int*, int*, double*, double*, int*,
   double*, int*);
3 extern void dgemm_(char*, char*, int*, int*, int*, double*, double*, int*, double*,
   int*, double*, double*, int*);
4
5 int trinvl_(char* diag, int* n, double* A, int* ldA, int* blocksize) {
6     if (*n == 1) {
7         if (diag[0] == 'N')
8             *A = 1 / *A;
9         return 0;
10    }
11
12    int ione = 1;
13    double one = 1;
14    double minusone = -1;
15
16    int p = 0;
17    while (p < *n) {
18        int b = *blocksize;
19        if (p + b > *n)
20            b = *n - p;
21        int r = *n - (p + b);
22
23        // A00
24        // A10 A11
25        // A20 A21 A22
```

```

26 #define A00 (A
27 #define A10 (A + p)
28 #define A11 (A + *ldA * p + p)
29 #define A20 (A + p + b)
30 #define A21 (A + *ldA * p + p + b)
31 #define A22 (A + *ldA * (p + b) + p + b)
32
33 // A10 = A10 * A00
34 dtrmm_("R", "L", "N", diag, &b, &p, &one, A00, ldA, A10, ldA);
35 // A10 = -A11 \ A10
36 dtrsm_("L", "L", "N", diag, &b, &p, &minusone, A11, ldA, A10, ldA);
37 // A11 = 1 / A11
38 trinv1_(diag, &b, A11, ldA, &ione);
39
40 p += b;
41 }
42 return 0;
43 }
44
45 int trinv2_(char* diag, int* n, double* A, int* ldA, int* blocksize) {
46     if (*n == 1) {
47         if (diag[0] == 'N')
48             *A = 1 / *A;
49         return 0;
50     }
51
52     int ione = 1;
53     double one = 1;
54     double minusone = -1;
55
56     int p = 0;
57     while (p < *n) {
58         int b = *blocksize;
59         if (p + b > *n)
60             b = *n - p;
61         int r = *n - (p + b);
62
63         // A00
64         // A10 A11
65         // A20 A21 A22
66 #define A00 (A
67 #define A10 (A + p)
68 #define A11 (A + *ldA * p + p)
69 #define A20 (A + p + b)
70 #define A21 (A + *ldA * p + p + b)
71 #define A22 (A + *ldA * (p + b) + p + b)
72
73         // A21 = A22 \ A21
74         dtrsm_("L", "L", "N", diag, &r, &b, &one, A22, ldA, A21, ldA);
75         // A21 = -A21 / A11
76         dtrsm_("R", "L", "N", diag, &r, &b, &minusone, A11, ldA, A21, ldA);
77         // A11 = 1 / A11
78         trinv2_(diag, &b, A11, ldA, &ione);
79
80         p += b;
81     }
82     return 0;
83 }
84
85 int trinv3_(char* diag, int* n, double* A, int* ldA, int* blocksize) {
86     if (*n == 1) {
87         if (diag[0] == 'N')

```

```

88         *A = 1 / *A;
89         return 0;
90     }
91
92     int ione = 1;
93     double one = 1;
94     double minusone = -1;
95
96     int p = 0;
97     while (p < *n) {
98         int b = *blocksize;
99         if (p + b > *n)
100             b = *n - p;
101         int r = *n - (p + b);
102
103         // A00
104         // A10 A11
105         // A20 A21 A22
106 #define A00 (A)
107 #define A10 (A + *ldA * p + p)
108 #define A11 (A + *ldA * p + p)
109 #define A20 (A + *ldA * p + p + b)
110 #define A21 (A + *ldA * p + p + b)
111 #define A22 (A + *ldA * (p + b) + p + b)
112
113         // A21 = -A21 / A11
114         dtrsm_("R", "L", "N", diag, &r, &b, &minusone, A11, ldA, A21, ldA);
115         // A20 = A21 * A10 + A20
116         dgemm_("N", "N", &r, &p, &b, &one, A21, ldA, A10, ldA, &one, A20, ldA);
117         // A10 = A11 \ A10
118         dtrsm_("L", "L", "N", diag, &b, &p, &one, A11, ldA, A10, ldA);
119         // A11 = 1 / A11
120         trinv3_(diag, &b, A11, ldA, &ione);
121
122         p += b;
123     }
124     return 0;
125 }
126
127 int trinv4_(char* diag, int* n, double* A, int* ldA, int* blocksize) {
128     if (*n == 1) {
129         if (diag[0] == 'N')
130             *A = 1 / *A;
131         return 0;
132     }
133
134     int ione = 1;
135     double one = 1;
136     double minusone = -1;
137
138     int p = 0;
139     while (p < *n) {
140         int b = *blocksize;
141         if (p + b > *n)
142             b = *n - p;
143         int r = *n - (p + b);
144
145         // A00
146         // A10 A11
147         // A20 A21 A22
148 #define A00 (A)
149 #define A10 (A + *ldA * p + p)

```

```

150 #define A11 (A + *ldA * p      + p)
151 #define A20 (A      + p + b)
152 #define A21 (A + *ldA * p      + p + b)
153 #define A22 (A + *ldA * (p + b) + p + b)
154
155 // A21 = -A22 \ A21
156 dtrsm_("L", "L", "N", diag, &r, &b, &minusone, A22, ldA, A21, ldA);
157 // A20 = -A21 * A10 + A20
158 dgemm_("N", "N", &r, &p, &b, &minusone, A21, ldA, A10, ldA, &one, A20, ldA)
159 ;
160 // A10 = A10 * A00
161 dtrmm_("R", "L", "N", diag, &b, &p, &one, A00, ldA, A10, ldA);
162 // A11 = 1 / A11
163 trinv4_(diag, &b, A11, ldA, &ione);
164
165 p += b;
166 }
167 return 0;
168 }

```

B.2 LU Decomposition $LU \leftarrow A$

Listing B.2: lu.c — LU decomposition.

```

1 extern void dtrmm_(char*, char*, char*, char*, int*, int*, double*, double*, int*,
2   double*, int*);
3 extern void dtrsm_(char*, char*, char*, char*, int*, int*, double*, double*, int*,
4   double*, int*);
5 extern void dgemm_(char*, char*, int*, int*, int*, double*, double*, int*, double*,
6   int*, double*, double*, int*);
7
8 int lu1_(int* n, double* A, int* ldA, int* blocksize) {
9   if (*n == 1)
10     return 0;
11
12   int ione = 1;
13   double one = 1;
14   double minusone = -1;
15
16   int p = 0;
17   while (p < *n) {
18     int b = *blocksize;
19     if (p + b > *n)
20       b = *n - p;
21     int r = *n - (p + b);
22
23     // A00 A01 A02
24     // A10 A11 A12
25     // A20 A21 A22
26 #define A00 (A)
27 #define A01 (A + *ldA * p)
28 #define A02 (A + *ldA * (p + b))
29 #define A10 (A      + p)
30 #define A11 (A + *ldA * p      + p)
31 #define A12 (A + *ldA * (p + b) + p)
32 #define A20 (A      + p + b)
33 #define A21 (A + *ldA * p      + p + b)
34 #define A22 (A + *ldA * (p + b) + p + b)

```



```

33 // A01 = trilu( A00 ) \ A01
34 dtrsm_( "L", "L", "N", "U", &p, &b, &one, A00, ldA, A01, ldA );
35 // A10 = A10 / triu( A00 )
36 dtrsm_( "R", "U", "N", "N", &b, &p, &one, A00, ldA, A10, ldA );
37 // A11 = A11 - A10 * A01
38 dgemm_( "N", "N", &b, &b, &p, &minusone, A10, ldA, A01, ldA, &one, A11, ldA )
39 ;
40 // A11 = LU(A11)
41 lu1_( &b, A11, ldA, &ione );
42
43 p += b;
44 }
45 return 0;
46 }
47
48 int lu2_( int* n, double* A, int* ldA, int* blocksize ) {
49     if ( *n == 1 )
50         return 0;
51
52     int ione = 1;
53     double one = 1;
54     double minusone = -1;
55
56     int p = 0;
57     while ( p < *n ) {
58         int b = *blocksize;
59         if ( p + b > *n )
60             b = *n - p;
61         int r = *n - ( p + b );
62
63         // A00 A01 A02
64         // A10 A11 A12
65         // A20 A21 A22
66 #define A00 (A)
67 #define A01 (A + *ldA * p)
68 #define A02 (A + *ldA * (p + b))
69 #define A10 (A + *ldA * p + p)
70 #define A11 (A + *ldA * p + p)
71 #define A12 (A + *ldA * (p + b) + p)
72 #define A20 (A + *ldA * p + p + b)
73 #define A21 (A + *ldA * p + p + b)
74 #define A22 (A + *ldA * (p + b) + p + b)
75
76 // A10 = A10 / triu( A00 )
77 dtrsm_( "R", "U", "N", "N", &b, &p, &one, A00, ldA, A10, ldA );
78 // A11 = A11 - A10 * A01
79 dgemm_( "N", "N", &b, &b, &p, &minusone, A10, ldA, A01, ldA, &one, A11, ldA )
80 ;
81 // A11 = LU(A11)
82 lu2_( &b, A11, ldA, &ione );
83 // A12 = A12 - A10 * A02
84 dgemm_( "N", "N", &b, &r, &p, &minusone, A10, ldA, A02, ldA, &one, A12, ldA )
85 ;
86 // A12 = trilu( A11 ) \ A12
87 dtrsm_( "L", "L", "N", "U", &b, &r, &one, A11, ldA, A12, ldA );
88
89 p += b;
90 }
91 return 0;
92 }
93
94 int lu3_( int* n, double* A, int* ldA, int* blocksize ) {

```

```

92     if (*n == 1)
93         return 0;
94
95     int ione = 1;
96     double one = 1;
97     double minusone = -1;
98
99     int p = 0;
100    while (p < *n) {
101        int b = *blocksize;
102        if (p + b > *n)
103            b = *n - p;
104        int r = *n - (p + b);
105
106        // A00 A01 A02
107        // A10 A11 A12
108        // A20 A21 A22
109    #define A00 (A)
110    #define A01 (A + *ldA * p)
111    #define A02 (A + *ldA * (p + b))
112    #define A10 (A + p)
113    #define A11 (A + *ldA * p + p)
114    #define A12 (A + *ldA * (p + b) + p)
115    #define A20 (A + p + b)
116    #define A21 (A + *ldA * p + p + b)
117    #define A22 (A + *ldA * (p + b) + p + b)
118
119
120        // A01 = trilu( A00 ) \ A01
121        dtrsm_( "L", "L", "N", "U", &p, &b, &one, A00, ldA, A01, ldA );
122        // A11 = A11 - A10 * A01
123        dgemm_( "N", "N", &b, &b, &p, &minusone, A10, ldA, A01, ldA, &one, A11, ldA )
124        ;
125        // A11 = LU(A11)
126        lu3_( &b, A11, ldA, &ione );
127        // A21 = A21 - A20 * A01
128        dgemm_( "N", "N", &r, &b, &p, &minusone, A20, ldA, A01, ldA, &one, A21, ldA )
129        ;
130        // A21 = A21 / triu( A11 )
131        dtrsm_( "R", "U", "N", "N", &r, &b, &one, A11, ldA, A21, ldA );
132
133        p += b;
134    }
135    return 0;
136 }
137
138 int lu4_(int* n, double* A, int* ldA, int* blocksize) {
139     if (*n == 1)
140         return 0;
141
142     int ione = 1;
143     double one = 1;
144     double minusone = -1;
145
146     int p = 0;
147     while (p < *n) {
148         int b = *blocksize;
149         if (p + b > *n)
150             b = *n - p;
151         int r = *n - (p + b);
152
153         // A00 A01 A02

```

```

152 // A10 A11 A12
153 // A20 A21 A22
154 #define A00 (A)
155 #define A01 (A + *ldA * p)
156 #define A02 (A + *ldA * (p + b))
157 #define A10 (A + p)
158 #define A11 (A + *ldA * p + p)
159 #define A12 (A + *ldA * (p + b) + p)
160 #define A20 (A + p + b)
161 #define A21 (A + *ldA * p + p + b)
162 #define A22 (A + *ldA * (p + b) + p + b)
163
164 // A11 = A11 - A10 * A01
165 dgemm_("N", "N", &b, &b, &p, &minusone, A10, ldA, A01, ldA, &one, A11, ldA)
166 ;
167 // A11 = LU(A11)
168 lu4_(&b, A11, ldA, &ione);
169 // A12 = A12 - A10 * A02
170 dgemm_("N", "N", &b, &r, &p, &minusone, A10, ldA, A02, ldA, &one, A12, ldA)
171 ;
172 // A12 = trilu( A11 ) \ A12
173 dtrsm_("L", "L", "N", "U", &b, &r, &one, A11, ldA, A12, ldA);
174 // A21 = A21 - A20 * A01
175 dgemm_("N", "N", &r, &b, &p, &minusone, A20, ldA, A01, ldA, &one, A21, ldA)
176 ;
177 // A21 = A21 / triu( A11 )
178 dtrsm_("R", "U", "N", "N", &r, &b, &one, A11, ldA, A21, ldA);
179
180 p += b;
181 }
182 return 0;
183 }
184
185 int lu5_(int* n, double* A, int* ldA, int* blocksize) {
186     if (*n == 1)
187         return 0;
188
189     int ione = 1;
190     double one = 1;
191     double minusone = -1;
192
193     int p = 0;
194     while (p < *n) {
195         int b = *blocksize;
196         if (p + b > *n)
197             b = *n - p;
198         int r = *n - (p + b);
199
200         // A00 A01 A02
201         // A10 A11 A12
202         // A20 A21 A22
203 #define A00 (A)
204 #define A01 (A + *ldA * p)
205 #define A02 (A + *ldA * (p + b))
206 #define A10 (A + p)
207 #define A11 (A + *ldA * p + p)
208 #define A12 (A + *ldA * (p + b) + p)
209 #define A20 (A + p + b)
210 #define A21 (A + *ldA * p + p + b)
211 #define A22 (A + *ldA * (p + b) + p + b)
212
213         // A11 = LU(A11)

```

```

211     lu5_(&b, A11, ldA, &ione);
212     // A12 = trilu( A11 ) \ A12
213     dtrsm_( "L", "L", "N", "U", &b, &r, &one, A11, ldA, A12, ldA);
214     // A21 = A21 / triu( A11 )
215     dtrsm_( "R", "U", "N", "N", &r, &b, &one, A11, ldA, A21, ldA);
216     // A22 = A22 - A21 * A12
217     dgemm_( "N", "N", &r, &r, &b, &minusone, A21, ldA, A12, ldA, &one, A22, ldA)
218         ;
219     p += b;
220 }
221 return 0;
222 }

```

B.3 Sylvester Equation: Solving $LX + XU = C$ for X

Since the file containing all 16 variants of the Sylvester Equation solver spans 1,127 lines, we only give the full code up to and including the first variant. For variants 2 through 16, we list only the update statements; the surrounding part of the algorithms is, up to the name, identical to the first variant.

Listing B.3: `sylv.c` (extract up to first algorithm) — Solution of Sylvester Equation.

```

1  extern void dgemm_(char*, char*, int*, int*, int*, double*, double*, int*, double*,
2      int*, double*, double*, int*);
3
4      // L00 L01 L02
5      // L10 L11 L12
6      // L20 L21 L22
7
8  #define L00m Lp
9  #define L01m Lp
10 #define L02m Lp
11 #define L10m Lb
12 #define L11m Lb
13 #define L12m Lb
14 #define L20m Lr
15 #define L21m Lr
16 #define L22m Lr
17
18 #define L00n Lp
19 #define L01n Lb
20 #define L02n Lr
21 #define L10n Lp
22 #define L11n Lb
23 #define L12n Lr
24 #define L20n Lp
25 #define L21n Lb
26 #define L22n Lr
27
28 #define L00 (L)
29 #define L01 (L + *ldL * Lp)
30 #define L02 (L + *ldL * (Lp + Lb))
31 #define L10 (L + *ldL * Lp + Lp)
32 #define L11 (L + *ldL * Lp + Lp)
33 #define L12 (L + *ldL * (Lp + Lb) + Lp)
34 #define L20 (L + *ldL * (Lp + Lb) + Lp + Lb)
35 #define L21 (L + *ldL * Lp + Lp + Lb)
36 #define L22 (L + *ldL * (Lp + Lb) + Lp + Lb)
37
38 // U00 U01 U02
39 // U10 U11 U12

```

```

36 // U20 U21 U22
37 #define U00m Up
38 #define U01m Up
39 #define U02m Up
40 #define U10m Ub
41 #define U11m Ub
42 #define U12m Ub
43 #define U20m Ur
44 #define U21m Ur
45 #define U22m Ur
46 #define U00n Up
47 #define U01n Ub
48 #define U02n Ur
49 #define U10n Up
50 #define U11n Ub
51 #define U12n Ur
52 #define U20n Up
53 #define U21n Ub
54 #define U22n Ur
55 #define U00 (U)
56 #define U01 (U + *ldU * Up)
57 #define U02 (U + *ldU * (Up + Ub))
58 #define U10 (U + *ldU * Up + Up)
59 #define U11 (U + *ldU * Up + Up)
60 #define U12 (U + *ldU * (Up + Ub) + Up)
61 #define U20 (U + *ldU * Up + Up + Ub)
62 #define U21 (U + *ldU * Up + Up + Ub)
63 #define U22 (U + *ldU * (Up + Ub) + Up + Ub)
64
65 // X00 X01 X02
66 // X10 X11 X12
67 // X20 X21 X22
68 #define X00m Lp
69 #define X01m Lp
70 #define X02m Lp
71 #define X10m Lb
72 #define X11m Lb
73 #define X12m Lb
74 #define X20m Lr
75 #define X21m Lr
76 #define X22m Lr
77 #define X00n Up
78 #define X01n Ub
79 #define X02n Ur
80 #define X10n Up
81 #define X11n Ub
82 #define X12n Ur
83 #define X20n Up
84 #define X21n Ub
85 #define X22n Ur
86 #define X00 (X)
87 #define X01 (X + *ldX * Up)
88 #define X02 (X + *ldX * (Up + Ub))
89 #define X10 (X + *ldX * Up + Lp)
90 #define X11 (X + *ldX * Up + Lp)
91 #define X12 (X + *ldX * (Up + Ub) + Lp)
92 #define X20 (X + *ldX * Up + Lp + Lb)
93 #define X21 (X + *ldX * Up + Lp + Lb)
94 #define X22 (X + *ldX * (Up + Ub) + Lp + Lb)
95
96
97 int sylv1_(int* m, int* n, double* L, int* ldL, double* U, int* ldU, double* X, int

```

```

    * ldX, int* blocksize) {
98 #undef sylv_
99 #define sylv_ sylv1_
100     if (*m * *n == 0)
101         return 0;
102     if (*m * *n == 1) {
103         *X /= *L + *U;
104         return 0;
105     }
106
107     int ione = 1;
108     double one = 1;
109     double mone = -1;
110
111     int p = 0;
112     int b = *blocksize;
113     if (b >= *m && b >= *n)
114         b = 1;
115     while (p < *n || p < *m) {
116
117         int Lp = p;
118         int Lb = b;
119         if (Lp >= *m) {
120             Lp = *m;
121             Lb = 0;
122         } else
123             if (Lp + Lb > *m)
124                 Lb = *m - Lp;
125         int Lr = *m - (Lp + Lb);
126
127         int Up = p;
128         int Ub = b;
129         if (Up >= *n) {
130             Up = *n;
131             Ub = 0;
132         } else
133             if (Up + Ub > *n)
134                 Ub = *n - Up;
135         int Ur = *n - (Up + Ub);
136
137         // X01 = -X00 U01 + X01
138         dgemm_("N", "N", &X01m, &X01n, &X00n, &mone, X00, ldX, U01, ldU, &one, X01,
139             ldX);
140         // X10 = -L10 X00 + X10
141         dgemm_("N", "N", &X10m, &X10n, &L10n, &mone, L10, ldL, X00, ldX, &one, X10,
142             ldX);
143         // X01 = sylv(L00, U11, X01)
144         sylv_(&X01m, &X01n, L00, ldL, U11, ldU, X01, ldX, blocksize);
145         // X10 = sylv(L11, U00, X10)
146         sylv_(&X10m, &X10n, L11, ldL, U00, ldU, X10, ldX, blocksize);
147         // X11 = -X10 U01 + X11
148         dgemm_("N", "N", &X11m, &X11n, &X10n, &mone, X10, ldX, U01, ldU, &one, X11,
149             ldX);
150         // X11 = -L10 X01 + X11
151         dgemm_("N", "N", &X11m, &X11n, &L10n, &mone, L10, ldL, X01, ldX, &one, X11,
152             ldX);
153         // X11 = sylv(L11, U11, X11)
154         sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
155
156         p += b;
157     }
158     return 0;

```

155 }

Listing B.4: `sylv.c` (2nd algorithm updates) — Solution of Sylvester Equation.

```

197 // X01 = -X00 U01 + X01
198 dgemm_("N", "N", &X01m, &X01n, &X00n, &mone, X00, ldX, U01, ldU, &one, X01, ldX);
199 // X10 = sylv(L11, U00, X10)
200 sylv_(&X10m, &X10n, L11, ldL, U00, ldU, X10, ldX, blocksize);
201 // X01 = sylv(L00, U11, X01)
202 sylv_(&X01m, &X01n, L00, ldL, U11, ldU, X01, ldX, blocksize);
203 // X11 = -X10 U01 + X11
204 dgemm_("N", "N", &X11m, &X11n, &X10n, &mone, X10, ldX, U01, ldU, &one, X11, ldX);
205 // X20 = -L21 X10 + X20
206 dgemm_("N", "N", &X20m, &X20n, &L21n, &mone, L21, ldL, X10, ldX, &one, X20, ldX);
207 // X11 = -L10 X01 + X11
208 dgemm_("N", "N", &X11m, &X11n, &L10n, &mone, L10, ldL, X01, ldX, &one, X11, ldX);
209 // X11 = sylv(L11, U11, X11)
210 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
211 // X21 = -L21 X11 + X21
212 dgemm_("N", "N", &X21m, &X21n, &L21n, &mone, L21, ldL, X11, ldX, &one, X21, ldX);
213 // X21 = -L20 X01 + X21
214 dgemm_("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);

```

Listing B.5: `sylv.c` (3rd algorithm updates) — Solution of Sylvester Equation.

```

261 // X01 = -X00 U01 + X01
262 dgemm_("N", "N", &X01m, &X01n, &X00n, &mone, X00, ldX, U01, ldU, &one, X01, ldX);
263 // X11 = -X10 U01 + X11
264 dgemm_("N", "N", &X11m, &X11n, &X10n, &mone, X10, ldX, U01, ldU, &one, X11, ldX);
265 // X21 = -X20 U01 + X21
266 dgemm_("N", "N", &X21m, &X21n, &X20n, &mone, X20, ldX, U01, ldU, &one, X21, ldX);
267 // X01 = sylv(L00, U11, X01)
268 sylv_(&X01m, &X01n, L00, ldL, U11, ldU, X01, ldX, blocksize);
269 // X11 = -L10 X01 + X11
270 dgemm_("N", "N", &X11m, &X11n, &L10n, &mone, L10, ldL, X01, ldX, &one, X11, ldX);
271 // X11 = sylv(L11, U11, X11)
272 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
273 // X21 = -L21 X11 + X21
274 dgemm_("N", "N", &X21m, &X21n, &L21n, &mone, L21, ldL, X11, ldX, &one, X21, ldX);
275 // X21 = -L20 X01 + X21
276 dgemm_("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);
277 // X21 = sylv(L22, U11, X21)
278 sylv_(&X21m, &X21n, L22, ldL, U11, ldU, X21, ldX, blocksize);

```

Listing B.6: `sylv.c` (4th algorithm updates) — Solution of Sylvester Equation.

```

325 // X01 = -X00 U01 + X01
326 dgemm_("N", "N", &X01m, &X01n, &X00n, &mone, X00, ldX, U01, ldU, &one, X01, ldX);
327 // X12 = -X10 U02 + X12
328 dgemm_("N", "N", &X12m, &X12n, &X10n, &mone, X10, ldX, U02, ldU, &one, X12, ldX);
329 // X01 = sylv(L00, U11, X01)
330 sylv_(&X01m, &X01n, L00, ldL, U11, ldU, X01, ldX, blocksize);
331 // X11 = -L10 X01 + X11
332 dgemm_("N", "N", &X11m, &X11n, &L10n, &mone, L10, ldL, X01, ldX, &one, X11, ldX);
333 // X11 = sylv(L11, U11, X11)
334 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
335 // X21 = -L21 X11 + X21
336 dgemm_("N", "N", &X21m, &X21n, &L21n, &mone, L21, ldL, X11, ldX, &one, X21, ldX);
337 // X21 = -L20 X01 + X21
338 dgemm_("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);

```

```

339 // X21 = sylv(L22, U11, X21)
340 sylv_(&X21m, &X21n, L22, ldL, U11, ldU, X21, ldX, blocksize);
341 // X22 = -X21 U12 + X22
342 dgemm_("N", "N", &X22m, &X22n, &X21n, &mone, X21, ldX, U12, ldU, &one, X22, ldX);

```

Listing B.7: `sylv.c` (5th algorithm updates) — Solution of Sylvester Equation.

```

389 // X01 = sylv(L00, U11, X01)
390 sylv_(&X01m, &X01n, L00, ldL, U11, ldU, X01, ldX, blocksize);
391 // X10 = -L10 X00 + X10
392 dgemm_("N", "N", &X10m, &X10n, &L10n, &mone, L10, ldL, X00, ldX, &one, X10, ldX);
393 // X02 = -X01 U12 + X02
394 dgemm_("N", "N", &X02m, &X02n, &X01n, &mone, X01, ldX, U12, ldU, &one, X02, ldX);
395 // X10 = sylv(L11, U00, X10)
396 sylv_(&X10m, &X10n, L11, ldL, U00, ldU, X10, ldX, blocksize);
397 // X11 = -X10 U01 + X11
398 dgemm_("N", "N", &X11m, &X11n, &X10n, &mone, X10, ldX, U01, ldU, &one, X11, ldX);
399 // X11 = -L10 X01 + X11
400 dgemm_("N", "N", &X11m, &X11n, &L10n, &mone, L10, ldL, X01, ldX, &one, X11, ldX);
401 // X11 = sylv(L11, U11, X11)
402 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
403 // X12 = -X11 U12 + X12
404 dgemm_("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
405 // X12 = -X10 U02 + X12
406 dgemm_("N", "N", &X12m, &X12n, &X10n, &mone, X10, ldX, U02, ldU, &one, X12, ldX);

```

Listing B.8: `sylv.c` (6th algorithm updates) — Solution of Sylvester Equation.

```

453 // X01 = sylv(L00, U11, X01)
454 sylv_(&X01m, &X01n, L00, ldL, U11, ldU, X01, ldX, blocksize);
455 // X10 = sylv(L11, U00, X10)
456 sylv_(&X10m, &X10n, L11, ldL, U00, ldU, X10, ldX, blocksize);
457 // X02 = -X01 U12 + X02
458 dgemm_("N", "N", &X02m, &X02n, &X01n, &mone, X01, ldX, U12, ldU, &one, X02, ldX);
459 // X11 = -X10 U01 + X11
460 dgemm_("N", "N", &X11m, &X11n, &X10n, &mone, X10, ldX, U01, ldU, &one, X11, ldX);
461 // X20 = -L21 X10 + X20
462 dgemm_("N", "N", &X20m, &X20n, &L21n, &mone, L21, ldL, X10, ldX, &one, X20, ldX);
463 // X11 = -L10 X01 + X11
464 dgemm_("N", "N", &X11m, &X11n, &L10n, &mone, L10, ldL, X01, ldX, &one, X11, ldX);
465 // X11 = sylv(L11, U11, X11)
466 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
467 // X12 = -X11 U12 + X12
468 dgemm_("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
469 // X21 = -L21 X11 + X21
470 dgemm_("N", "N", &X21m, &X21n, &L21n, &mone, L21, ldL, X11, ldX, &one, X21, ldX);
471 // X12 = -X10 U02 + X12
472 dgemm_("N", "N", &X12m, &X12n, &X10n, &mone, X10, ldX, U02, ldU, &one, X12, ldX);
473 // X21 = -L20 X01 + X21
474 dgemm_("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);

```

Listing B.9: `sylv.c` (7th algorithm updates) — Solution of Sylvester Equation.

```

521 // X01 = sylv(L00, U11, X01)
522 sylv_(&X01m, &X01n, L00, ldL, U11, ldU, X01, ldX, blocksize);
523 // X11 = -X10 U01 + X11
524 dgemm_("N", "N", &X11m, &X11n, &X10n, &mone, X10, ldX, U01, ldU, &one, X11, ldX);
525 // X21 = -X20 U01 + X21
526 dgemm_("N", "N", &X21m, &X21n, &X20n, &mone, X20, ldX, U01, ldU, &one, X21, ldX);
527 // X02 = -X01 U12 + X02

```



```

528 dgemm_("N", "N", &X02m, &X02n, &X01n, &mone, X01, ldX, U12, ldU, &one, X02, ldX);
529 // X11 = -L10 X01 + X11
530 dgemm_("N", "N", &X11m, &X11n, &L10n, &mone, L10, ldL, X01, ldX, &one, X11, ldX);
531 // X11 = sylv(L11, U11, X11)
532 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
533 // X12 = -X11 U12 + X12
534 dgemm_("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
535 // X21 = -L21 X11 + X21
536 dgemm_("N", "N", &X21m, &X21n, &L21n, &mone, L21, ldL, X11, ldX, &one, X21, ldX);
537 // X12 = -X10 U02 + X12
538 dgemm_("N", "N", &X12m, &X12n, &X10n, &mone, X10, ldX, U02, ldU, &one, X12, ldX);
539 // X21 = -L20 X01 + X21
540 dgemm_("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);
541 // X21 = sylv(L22, U11, X21)
542 sylv_(&X21m, &X21n, L22, ldL, U11, ldU, X21, ldX, blocksize);

```

Listing B.10: `sylv.c` (8th algorithm updates) — Solution of Sylvester Equation.

```

589 // X01 = sylv(L00, U11, X01)
590 sylv_(&X01m, &X01n, L00, ldL, U11, ldU, X01, ldX, blocksize);
591 // X02 = -X01 U12 + X02
592 dgemm_("N", "N", &X02m, &X02n, &X01n, &mone, X01, ldX, U12, ldU, &one, X02, ldX);
593 // X11 = -L10 X01 + X11
594 dgemm_("N", "N", &X11m, &X11n, &L10n, &mone, L10, ldL, X01, ldX, &one, X11, ldX);
595 // X11 = sylv(L11, U11, X11)
596 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
597 // X12 = -X11 U12 + X12
598 dgemm_("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
599 // X21 = -L21 X11 + X21
600 dgemm_("N", "N", &X21m, &X21n, &L21n, &mone, L21, ldL, X11, ldX, &one, X21, ldX);
601 // X21 = -L20 X01 + X21
602 dgemm_("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);
603 // X21 = sylv(L22, U11, X21)
604 sylv_(&X21m, &X21n, L22, ldL, U11, ldU, X21, ldX, blocksize);
605 // X22 = -X21 U12 + X22
606 dgemm_("N", "N", &X22m, &X22n, &X21n, &mone, X21, ldX, U12, ldU, &one, X22, ldX);

```

Listing B.11: `sylv.c` (9th algorithm updates) — Solution of Sylvester Equation.

```

653 // X01 = sylv(L00, U11, X01)
654 sylv_(&X01m, &X01n, L00, ldL, U11, ldU, X01, ldX, blocksize);
655 // X02 = -X01 U12 + X02
656 dgemm_("N", "N", &X02m, &X02n, &X01n, &mone, X01, ldX, U12, ldU, &one, X02, ldX);
657 // X11 = -L10 X01 + X11
658 dgemm_("N", "N", &X11m, &X11n, &L10n, &mone, L10, ldL, X01, ldX, &one, X11, ldX);
659 // X11 = sylv(L11, U11, X11)
660 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
661 // X12 = -X11 U12 + X12
662 dgemm_("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
663 // X21 = -L21 X11 + X21
664 dgemm_("N", "N", &X21m, &X21n, &L21n, &mone, L21, ldL, X11, ldX, &one, X21, ldX);
665 // X21 = -L20 X01 + X21
666 dgemm_("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);
667 // X21 = sylv(L22, U11, X21)
668 sylv_(&X21m, &X21n, L22, ldL, U11, ldU, X21, ldX, blocksize);
669 // X22 = -X21 U12 + X22
670 dgemm_("N", "N", &X22m, &X22n, &X21n, &mone, X21, ldX, U12, ldU, &one, X22, ldX);

```

Listing B.12: `sylv.c` (10th algorithm updates) — Solution of Sylvester Equation.

```

717 // X10 = -L10 X00 + X10
718 dgemm_ ("N", "N", &X10m, &X10n, &L10n, &mone, L10, ldL, X00, ldX, &one, X10, ldX);
719 // X21 = -L20 X01 + X21
720 dgemm_ ("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);
721 // X10 = sylv(L11, U00, X10)
722 sylv_(&X10m, &X10n, L11, ldL, U00, ldU, X10, ldX, blocksize);
723 // X11 = -X10 U01 + X11
724 dgemm_ ("N", "N", &X11m, &X11n, &X10n, &mone, X10, ldX, U01, ldU, &one, X11, ldX);
725 // X11 = sylv(L11, U11, X11)
726 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
727 // X12 = -X11 U12 + X12
728 dgemm_ ("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
729 // X12 = -X10 U02 + X12
730 dgemm_ ("N", "N", &X12m, &X12n, &X10n, &mone, X10, ldX, U02, ldU, &one, X12, ldX);
731 // X12 = sylv(L11, U22, X12)
732 sylv_(&X12m, &X12n, L11, ldL, U22, ldU, X12, ldX, blocksize);
733 // X22 = -L21 X12 + X22
734 dgemm_ ("N", "N", &X22m, &X22n, &L21n, &mone, L21, ldL, X12, ldX, &one, X22, ldX);

```

Listing B.13: sylv.c (11th algorithm updates) — Solution of Sylvester Equation.

```

781 // X10 = -L10 X00 + X10
782 dgemm_ ("N", "N", &X10m, &X10n, &L10n, &mone, L10, ldL, X00, ldX, &one, X10, ldX);
783 // X21 = -L20 X01 + X21
784 dgemm_ ("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);
785 // X10 = sylv(L11, U00, X10)
786 sylv_(&X10m, &X10n, L11, ldL, U00, ldU, X10, ldX, blocksize);
787 // X11 = -X10 U01 + X11
788 dgemm_ ("N", "N", &X11m, &X11n, &X10n, &mone, X10, ldX, U01, ldU, &one, X11, ldX);
789 // X11 = sylv(L11, U11, X11)
790 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
791 // X12 = -X11 U12 + X12
792 dgemm_ ("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
793 // X12 = -X10 U02 + X12
794 dgemm_ ("N", "N", &X12m, &X12n, &X10n, &mone, X10, ldX, U02, ldU, &one, X12, ldX);
795 // X12 = sylv(L11, U22, X12)
796 sylv_(&X12m, &X12n, L11, ldL, U22, ldU, X12, ldX, blocksize);
797 // X22 = -L21 X12 + X22
798 dgemm_ ("N", "N", &X22m, &X22n, &L21n, &mone, L21, ldL, X12, ldX, &one, X22, ldX);

```

Listing B.14: sylv.c (12th algorithm updates) — Solution of Sylvester Equation.

```

849 // X10 = -L10 X00 + X10
850 dgemm_ ("N", "N", &X10m, &X10n, &L10n, &mone, L10, ldL, X00, ldX, &one, X10, ldX);
851 // X21 = -L20 X01 + X21
852 dgemm_ ("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);
853 // X10 = sylv(L11, U00, X10)
854 sylv_(&X10m, &X10n, L11, ldL, U00, ldU, X10, ldX, blocksize);
855 // X11 = -X10 U01 + X11
856 dgemm_ ("N", "N", &X11m, &X11n, &X10n, &mone, X10, ldX, U01, ldU, &one, X11, ldX);
857 // X11 = sylv(L11, U11, X11)
858 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
859 // X12 = -X11 U12 + X12
860 dgemm_ ("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
861 // X12 = -X10 U02 + X12
862 dgemm_ ("N", "N", &X12m, &X12n, &X10n, &mone, X10, ldX, U02, ldU, &one, X12, ldX);
863 // X12 = sylv(L11, U22, X12)
864 sylv_(&X12m, &X12n, L11, ldL, U22, ldU, X12, ldX, blocksize);
865 // X22 = -L21 X12 + X22
866 dgemm_ ("N", "N", &X22m, &X22n, &L21n, &mone, L21, ldL, X12, ldX, &one, X22, ldX);

```

Listing B.15: `sylv.c` (13th algorithm updates) — Solution of Sylvester Equation.

```

913 // X11 = -X10 U01 + X11
914 dgemm_("N", "N", &X11m, &X11n, &X10n, &mone, X10, ldX, U01, ldU, &one, X11, ldX);
915 // X21 = -X20 U01 + X21
916 dgemm_("N", "N", &X21m, &X21n, &X20n, &mone, X20, ldX, U01, ldU, &one, X21, ldX);
917 // X11 = -L10 X01 + X11
918 dgemm_("N", "N", &X11m, &X11n, &L10n, &mone, L10, ldL, X01, ldX, &one, X11, ldX);
919 // X11 = sylv(L11, U11, X11)
920 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
921 // X12 = -X11 U12 + X12
922 dgemm_("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
923 // X21 = -L21 X11 + X21
924 dgemm_("N", "N", &X21m, &X21n, &L21n, &mone, L21, ldL, X11, ldX, &one, X21, ldX);
925 // X12 = -X10 U02 + X12
926 dgemm_("N", "N", &X12m, &X12n, &X10n, &mone, X10, ldX, U02, ldU, &one, X12, ldX);
927 // X21 = -L20 X01 + X21
928 dgemm_("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);
929 // X12 = -L10 X02 + X12
930 dgemm_("N", "N", &X12m, &X12n, &L10n, &mone, L10, ldL, X02, ldX, &one, X12, ldX);
931 // X21 = sylv(L22, U11, X21)
932 sylv_(&X21m, &X21n, L22, ldL, U11, ldU, X21, ldX, blocksize);
933 // X12 = sylv(L11, U22, X12)
934 sylv_(&X12m, &X12n, L11, ldL, U22, ldU, X12, ldX, blocksize);

```

Listing B.16: `sylv.c` (14th algorithm updates) — Solution of Sylvester Equation.

```

981 // X11 = -X10 U01 + X11
982 dgemm_("N", "N", &X11m, &X11n, &X10n, &mone, X10, ldX, U01, ldU, &one, X11, ldX);
983 // X21 = -X20 U01 + X21
984 dgemm_("N", "N", &X21m, &X21n, &X20n, &mone, X20, ldX, U01, ldU, &one, X21, ldX);
985 // X11 = sylv(L11, U11, X11)
986 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
987 // X12 = -X11 U12 + X12
988 dgemm_("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
989 // X21 = -L21 X11 + X21
990 dgemm_("N", "N", &X21m, &X21n, &L21n, &mone, L21, ldL, X11, ldX, &one, X21, ldX);
991 // X12 = -X10 U02 + X12
992 dgemm_("N", "N", &X12m, &X12n, &X10n, &mone, X10, ldX, U02, ldU, &one, X12, ldX);
993 // X21 = sylv(L22, U11, X21)
994 sylv_(&X21m, &X21n, L22, ldL, U11, ldU, X21, ldX, blocksize);
995 // X12 = sylv(L11, U22, X12)
996 sylv_(&X12m, &X12n, L11, ldL, U22, ldU, X12, ldX, blocksize);
997 // X22 = -L21 X12 + X22
998 dgemm_("N", "N", &X22m, &X22n, &L21n, &mone, L21, ldL, X12, ldX, &one, X22, ldX);

```

Listing B.17: `sylv.c` (15th algorithm updates) — Solution of Sylvester Equation.

```

1045 // X11 = -L10 X01 + X11
1046 dgemm_("N", "N", &X11m, &X11n, &L10n, &mone, L10, ldL, X01, ldX, &one, X11, ldX);
1047 // X11 = sylv(L11, U11, X11)
1048 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
1049 // X12 = -X11 U12 + X12
1050 dgemm_("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
1051 // X21 = -L21 X11 + X21
1052 dgemm_("N", "N", &X21m, &X21n, &L21n, &mone, L21, ldL, X11, ldX, &one, X21, ldX);
1053 // X12 = -L10 X02 + X12
1054 dgemm_("N", "N", &X12m, &X12n, &L10n, &mone, L10, ldL, X02, ldX, &one, X12, ldX);
1055 // X21 = -L20 X01 + X21
1056 dgemm_("N", "N", &X21m, &X21n, &L20n, &mone, L20, ldL, X01, ldX, &one, X21, ldX);
1057 // X12 = sylv(L11, U22, X12)

```

```

1058 sylv_(&X12m, &X12n, L11, ldL, U22, ldU, X12, ldX, blocksize);
1059 // X21 = sylv(L22, U11, X21)
1060 sylv_(&X21m, &X21n, L22, ldL, U11, ldU, X21, ldX, blocksize);
1061 // X22 = -X21 U12 + X22
1062 dgemm_("N", "N", &X22m, &X22n, &X21n, &mone, X21, ldX, U12, ldU, &one, X22, ldX);

```

Listing B.18: `sylv.c` (16th algorithm updates) — Solution of Sylvester Equation.

```

1109 // X11 = sylv(L11, U11, X11)
1110 sylv_(&X11m, &X11n, L11, ldL, U11, ldU, X11, ldX, blocksize);
1111 // X12 = -X11 U12 + X12
1112 dgemm_("N", "N", &X12m, &X12n, &X11n, &mone, X11, ldX, U12, ldU, &one, X12, ldX);
1113 // X21 = -L21 X11 + X21
1114 dgemm_("N", "N", &X21m, &X21n, &L21n, &mone, L21, ldL, X11, ldX, &one, X21, ldX);
1115 // X12 = sylv(L11, U22, X12)
1116 sylv_(&X12m, &X12n, L11, ldL, U22, ldU, X12, ldX, blocksize);
1117 // X21 = sylv(L22, U11, X21)
1118 sylv_(&X21m, &X21n, L22, ldL, U11, ldU, X21, ldX, blocksize);
1119 // X22 = -X21 U12 + X22
1120 dgemm_("N", "N", &X22m, &X22n, &X21n, &mone, X21, ldX, U12, ldU, &one, X22, ldX);
1121 // X22 = -L21 X12 + X22
1122 dgemm_("N", "N", &X22m, &X22n, &L21n, &mone, L21, ldL, X12, ldX, &one, X22, ldX);

```